Chapter 15 Query Proceesing

Query Process/ Optimization

- Study how database process query internally
  - i.e. Once a database system received an SQL query, what happens (until the database return the results)
- Why study?
  - Database project manager – understand whether the queries being written will likely be executed effectively by the database

    > Maybe I could break it down into multiple queries. All maybe this question is just simply not a good fit with the database system.

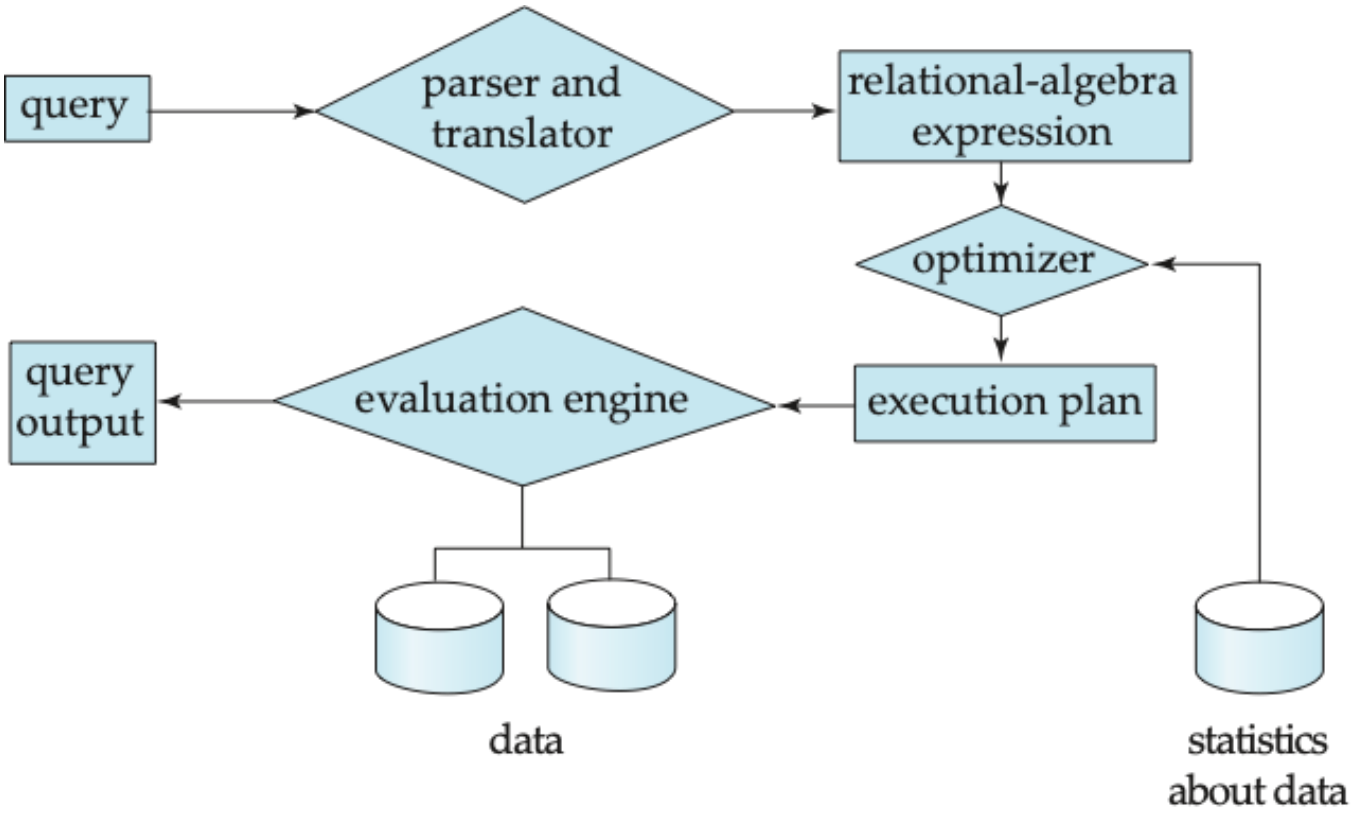
  - Database administrator – able to restructure database / provide hints to the database system to speed up queries

    > You'd only tell the database what you want. The database itself have to figure out how to get it right. And remember the earlier example, we talk about using an index. Doesn't mean you should use it. Remember the fundamental problem. The chicken and egg problem was the chicken and egg problem. Which came first? You need to know the answer the size of your answer to the query to figure out the best way to execute a query. But you cannot get the size of the result until you actually execute a query. So you have to decide what to do before, you know, a crucial piece of information so that you can do the best you
    >
    > You clearly have returned to two tuples. By all means, you use a non cluster index.
    >
    > 2000 tuples --> index.
    >
    > But you don't know about the result whether is 2 tuples or 2000 tuples.
    >
    > But in order for you to provide hints effectively, you need to understand how things work underneath.

  - Database developer – you may be hired by a database systems company to build the next version of a DBMS

Query Processing

- What happen when a DBMS received a query

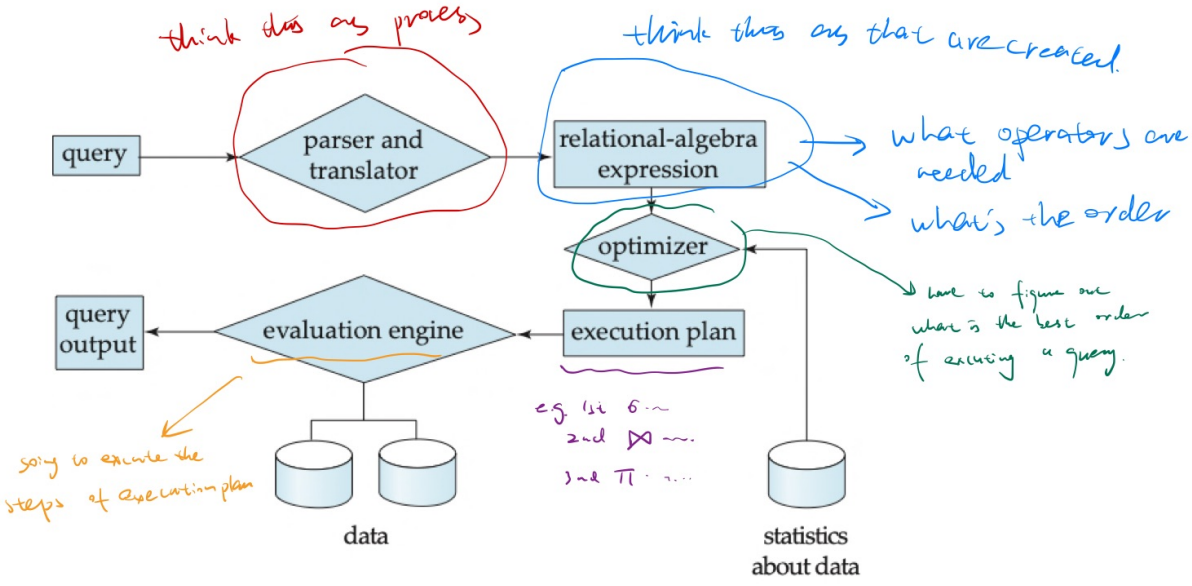


So when you receive a query, it will first go for a parser and translator. The goal is to convert your SQL query into a bunch of relational algebra operators.

Because the database engine works best with a set of relational algebra operators. make the task of developing a delivery system more modular and more easy to handle. And that makes the optimization job better.

algorithm select

Relational-algebra expression: Notice that at this date it typically I did give you a very preliminary list of those.



For example, I want to do selection on a table. Then I can now ask 2 questions. There are maybe 10 algorithms you can do to selection. And also, depends on whether you have an index or not. Or you have many index, which index should I use?

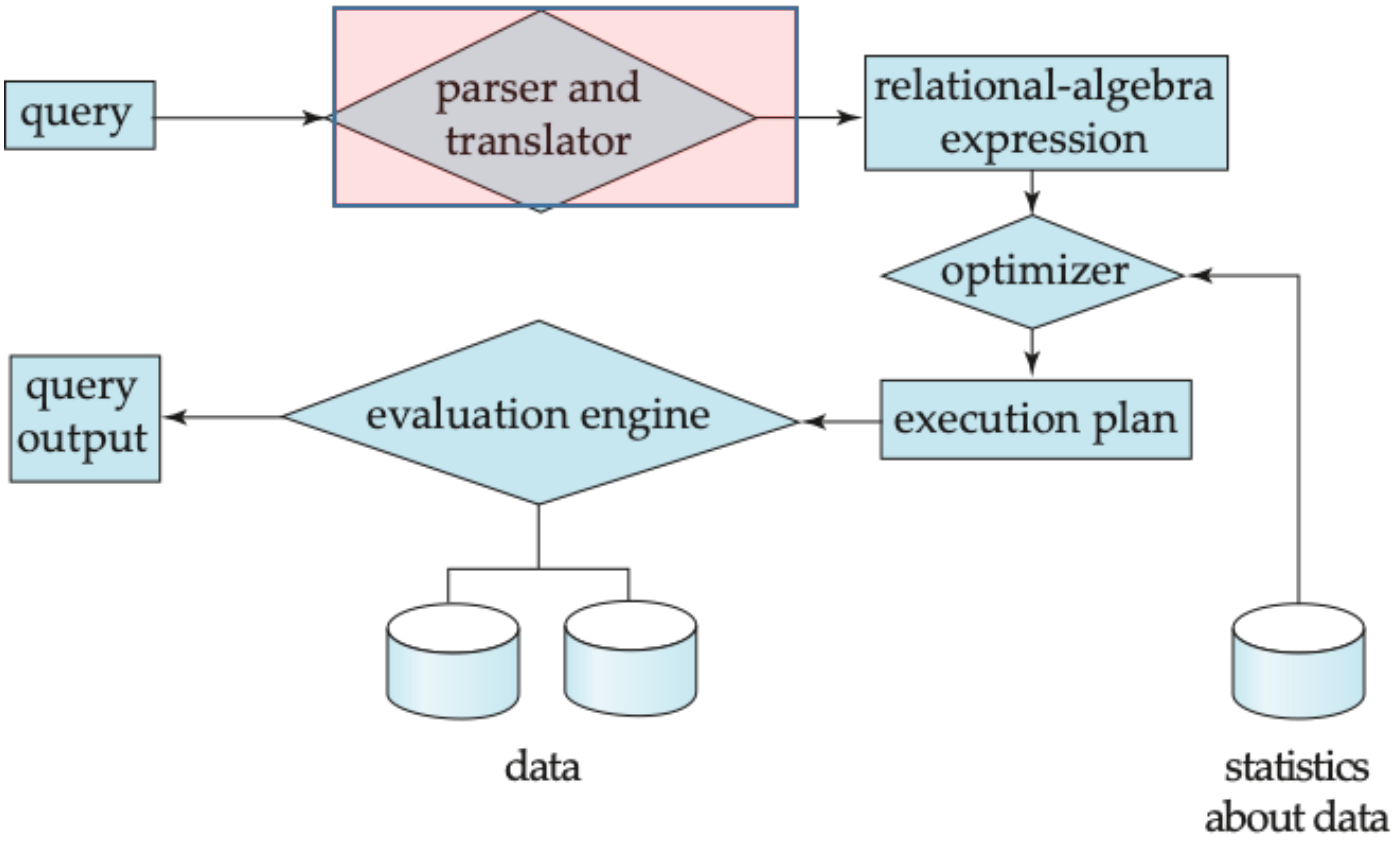
So, we need to figure out what is the best method of doing each operation. And then we have to figure out which operations should I do first?

Remember Join are associated. It doesn't matter the order.

$3 \times 4 \times 5$ I don't care you do 3×4 first or 4×5 first.

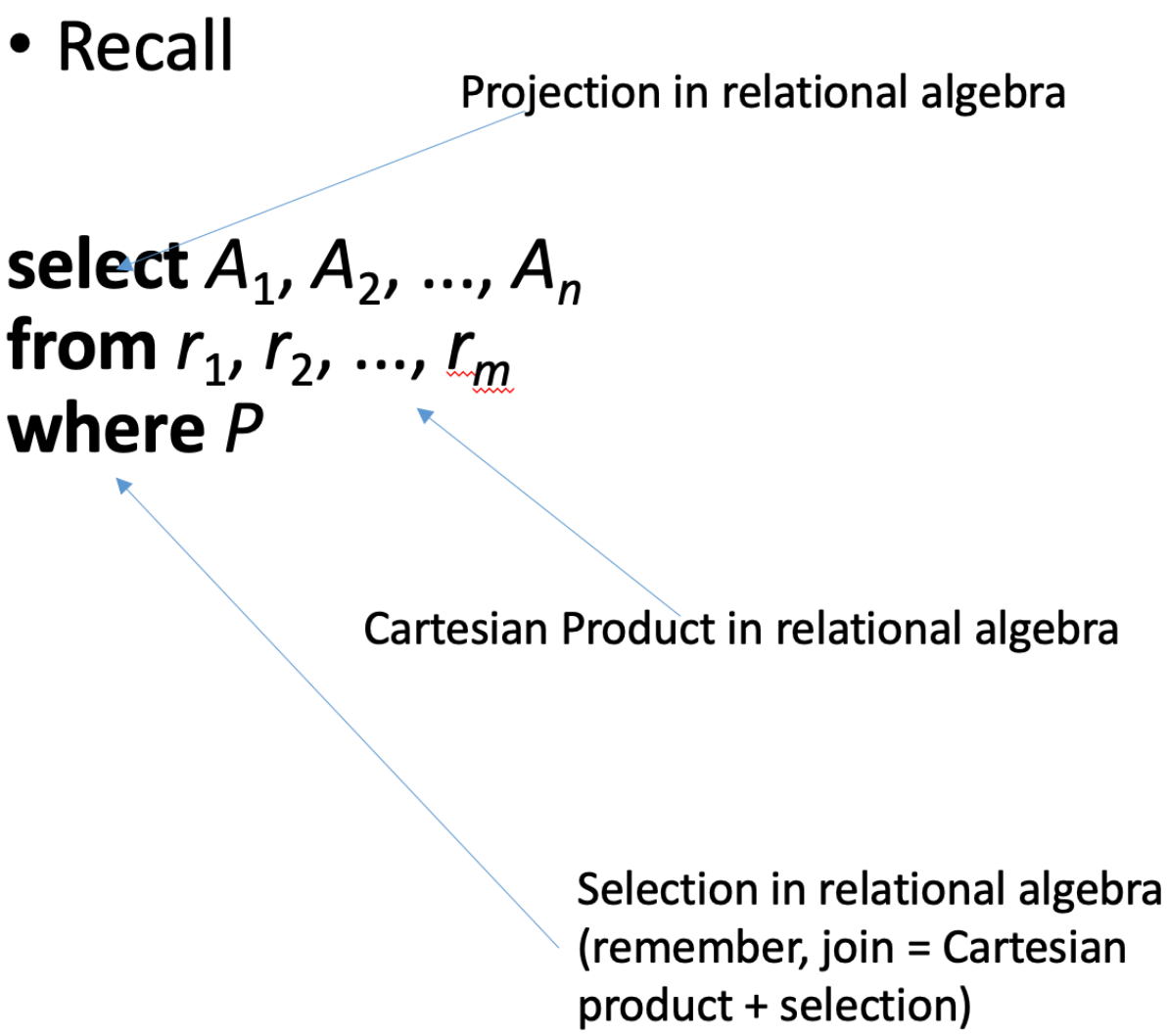
Parser and translator

- Parser and translator
- Read the query
- Check for syntax
- Collect all relevant information about tables involved
- Break the query down into a set of basic operations (relational algebra + others)



> This step are quickly step. It just tell you, hey, you need to do these things. You need to do that thing. They didn't tell you what should be the best order. They didn't tell you that what is the best algorithm to do each of these? They just tell you you have to do this.

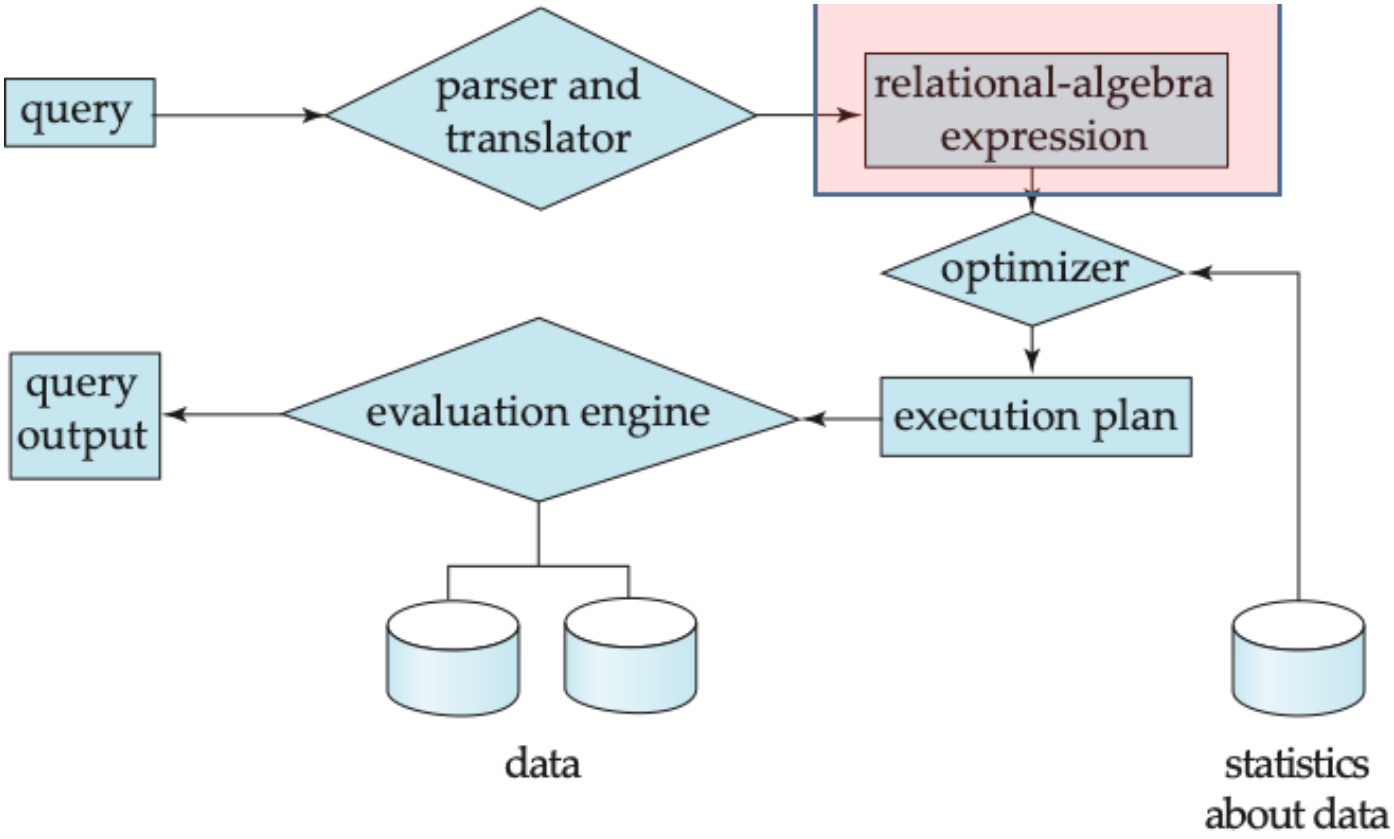
- Recall



> `from` : is essentially Cartesian product.
>
> `where` : is basically selection.
>
> `select` : is projection

- Query is converted to a list of operators ($\sigma, \prod, \bowtie, Group, Order, \ldots$)
- Thus running the query becomes execution of a list of operators

> running a query equivalent to executing a list of operate, and once again, that's why you learn relational algebra not necessarily to write a relational algebra query, because you don't have to do it in real life. But you understand how a query be executed.



- Example

```
SELECT ssn, age           // And then all I need is to attribute, so I need to project
FROM Instructor
WHERE age > 25          // this is a selection
```

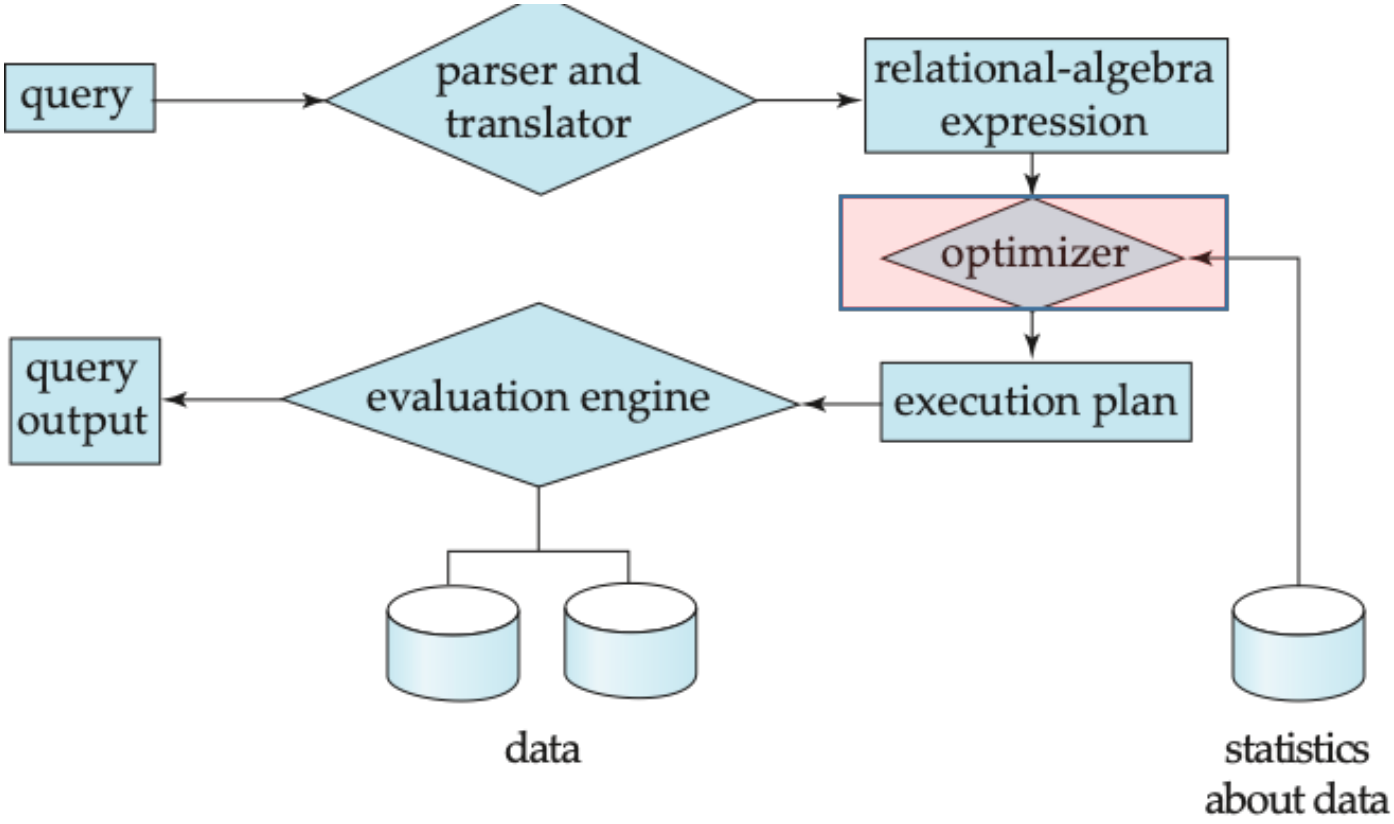
Becomes

1. $A = \sigma_{sage>25}(Instructor)$
2. $Result = \prod_{ssn,age}(A)$

> So for the parser and translator you stop here. I don't it this not necessarily the ideal order. They definitely don't tell you which algorithm you should use.

Optimizer

- For each operation, determine how it will be executed
- Determine the order of operations
- Other tasks (to be discussed later)



Execution plan

Example:

- 1. $A = \sigma_{age>25}(Instructor)$
  - Use the secondary index on A to retrieve the tuples
  - Do not store A on to the disk

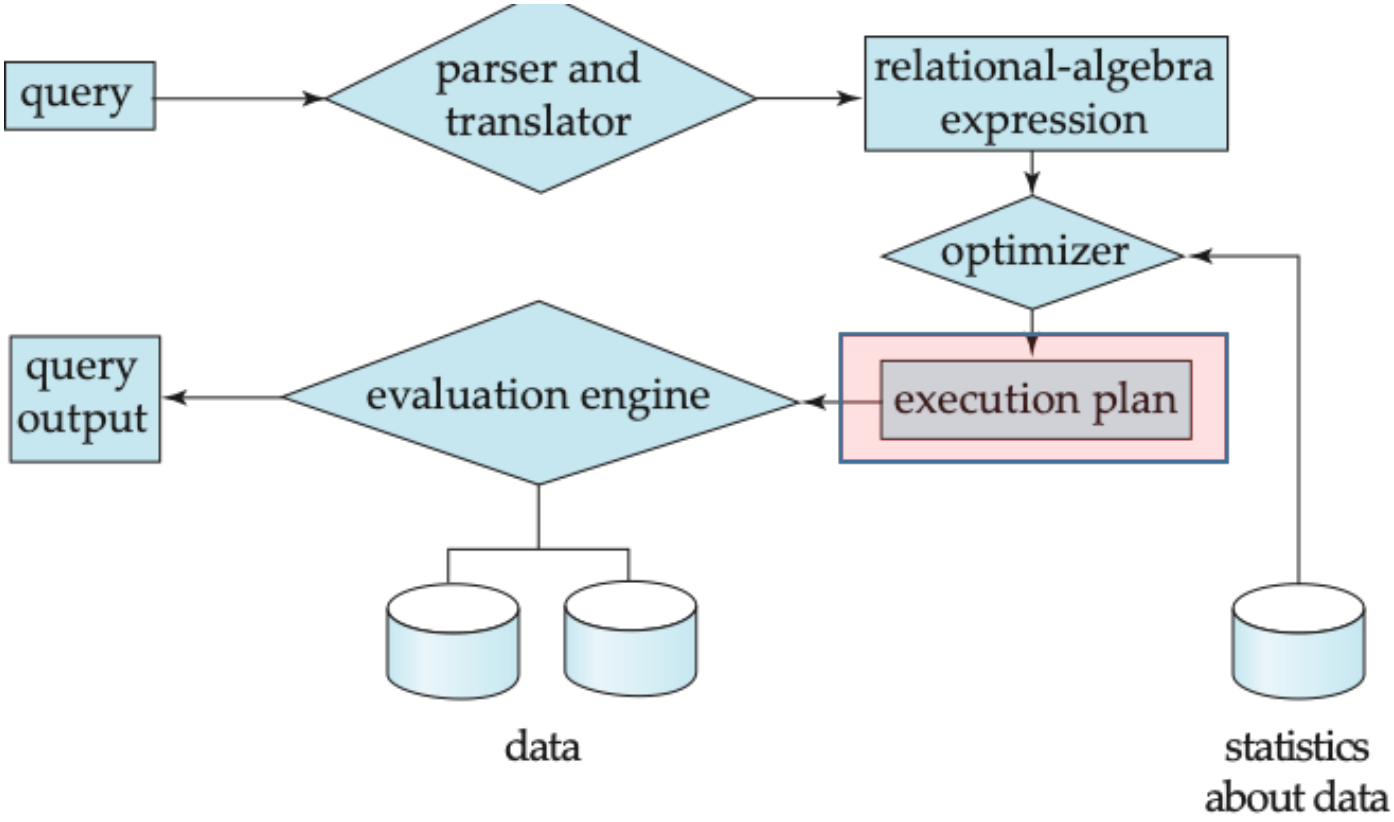
  > We will do this step first. Let's say that system said they say index on on each instructor. It is actually worth it to use it in this case and do not store it onto the disk.

- $Result = \prod_{ssn,age(A)}$
  - Directly pipe the result from 1 to execute the project

    > What do you mean by pipe? We'll talk about it later in the later in probably the weeks.

  - Just pick the attributes and output them

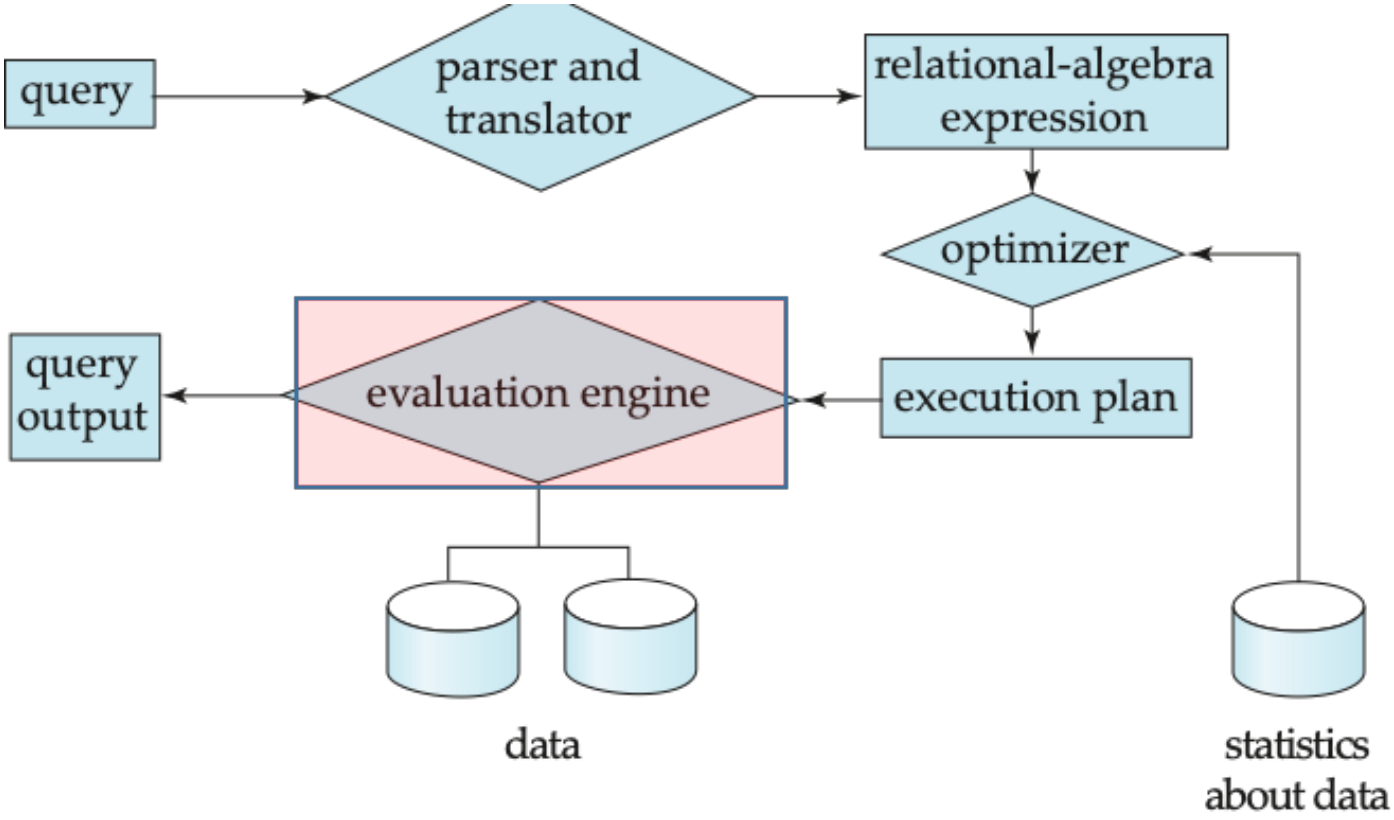
  > These two will be extra infromation decided by the optimizer.



Evaluation engine

Exvaluation engine

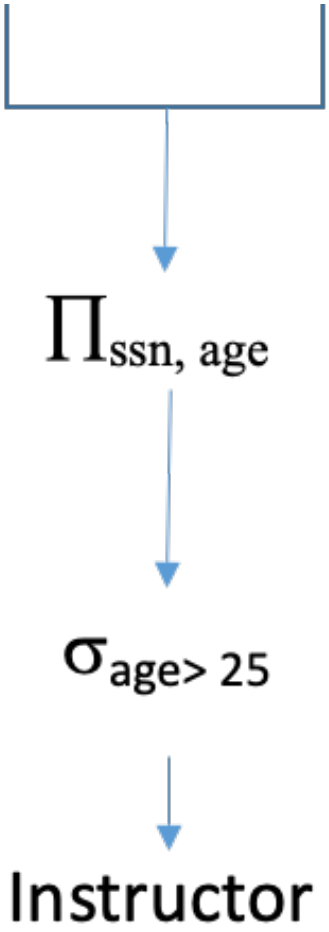
- Actual execute the plan
- Return the tuples to the output
    - Can be on screen, as a file, or as a stream through a network



> Now the executinon engine have all the information very specific enough they don't leave it to ask questions, just execute it.
>
> we will spend a lot of time looking at `parser and translator` and some time lookting at `optimizer` and we will also spend a lot of time at `evaluation engine`.

- Notice that more often, a query can be represented as a parse tree:

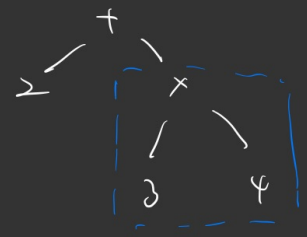
```
SELECT ssn, age
FROM Instructor
WHERE age > 25
```


$\Pi_{ssn,\ age}$
$\sigma_{age>25}$
Instructor

In data structure are a very common exercise.

In data structure to convert into a tree.

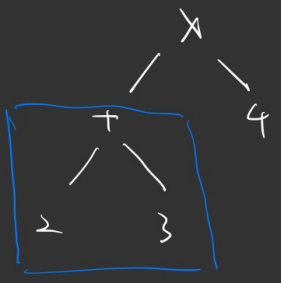
$$2 + 3 \times 4$$



remember going to the excute from the leafs up to the root

I multiply 3 and 4 and the I will add the result to 2.

If i write the tree on the other way the meaning change completely.



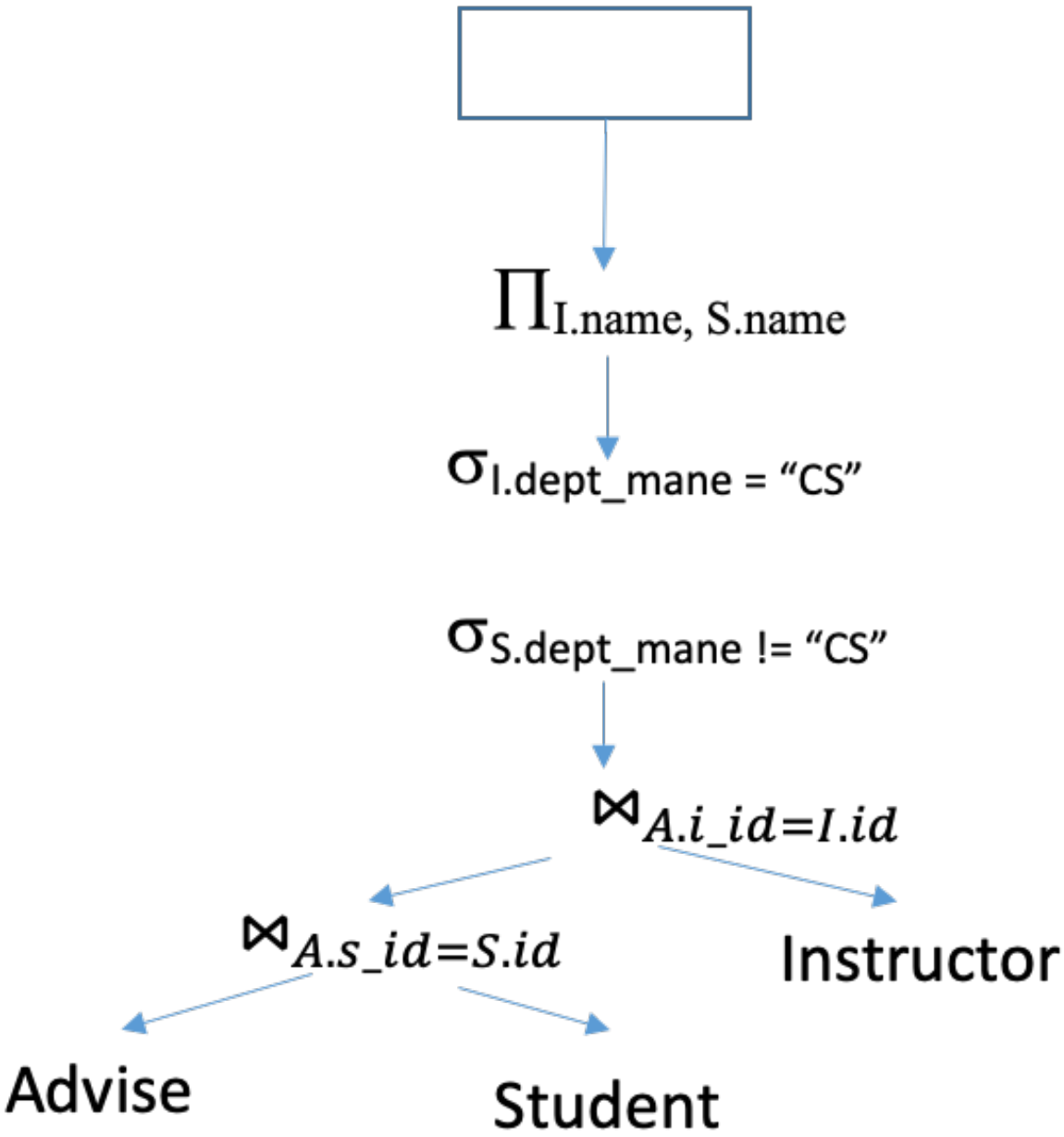
add 3+4 first then × 4

$$(2+3) \times 4$$

So what is the difference between the tree we plant and the tree that actually got?

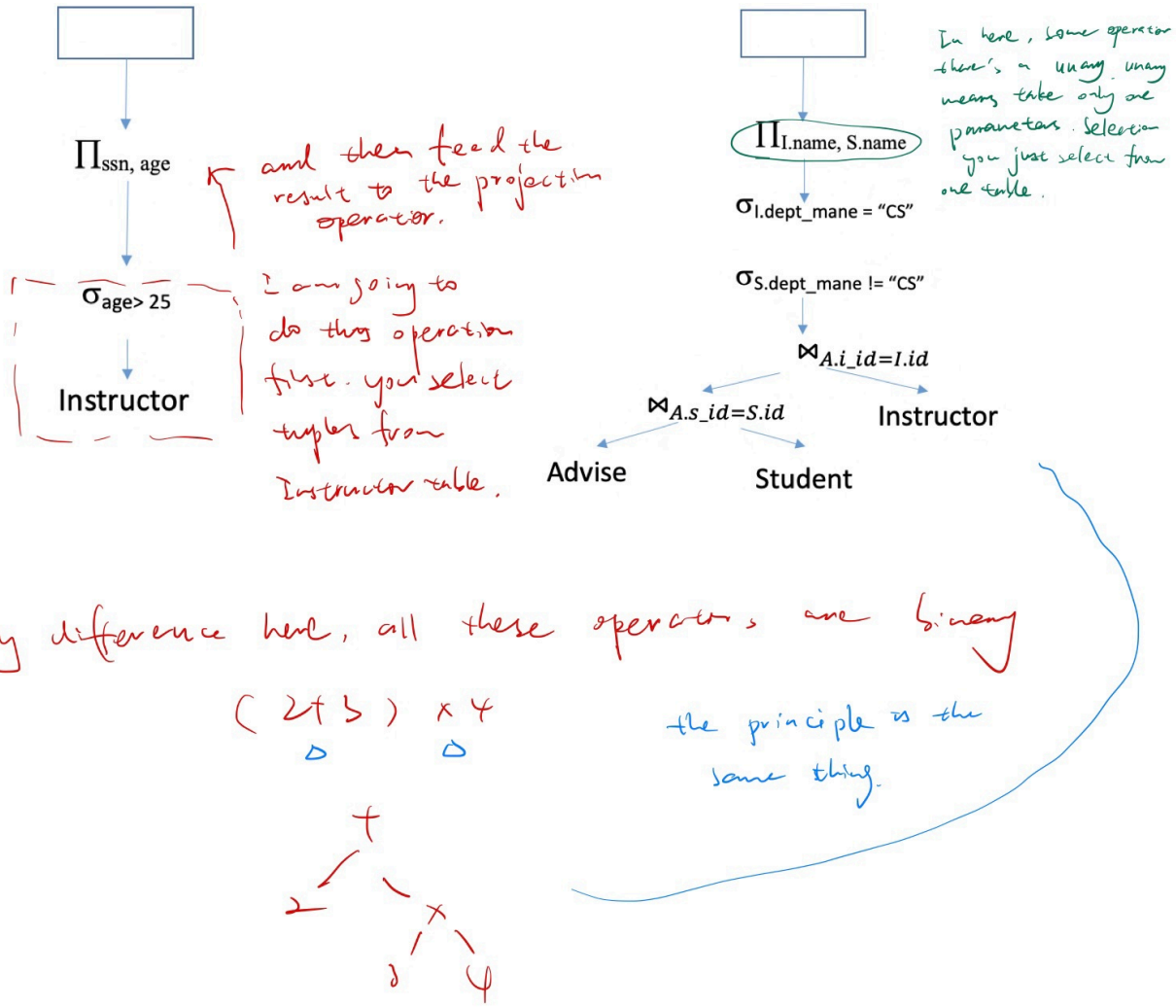
our tree goes downwards. except Bt tree. still upwards.

```
SELECT I.name, S.name
FROM Instructor I, Advise A, Student S
WHERE A.i_id = I.ID AND S.id = A.s_id
AND I.dept_name = “CS”
AND S.dept_name != “CS”
```



> So very often, at least conceptually, it is not a bad idea to represent a query as a tree

Very often, at least conceptually. It's not a bad idea to represent a query as a tree.



- Query Processing/Optimization in terms of parse trees:
  - Parsing: Generate parse trees for the query
    - Notice that a query may generate MANY (TOO MANY) valid parse trees

> join - associative
>
> selection - commutative - combine
>
> Projection - cometative - combine

> Projection - commutative - combine
>
> selection / projection - may be commutative
>
> So that means. There are many, potentially many potential risks.
>
> > for example $A \bowtie B \bowtie C$
> >
> > I don't care which of John you do first, you are going to get this.
> >
> > So, once again, the same chicken and egg problem again. You need to figure out the best way to do join. You need to know how many pupils are you to return for each join. But you don't exactly know this until you've executed the query.

    - Each node correspond to an operation
    - Evaluation of query is from bottom to top (think post order)
- Optimization:
    - Decorating the nodes of a parse
        - For each node, determine how that operation is to be executed
    - Select the best parse tree
        - For all the parse tree generated, pick the one that will execute the query fastest

> Just because you have an index doesn't mean you have to use it. Does that mean it's worth it to use it? To make it even stronger.

- Challenges
    - Combinatorics explosion

> combinatorics [kəm,bainə'tɔ:riks]
>
> n. [数]组合学；组合数学（等于 combinatorial analysis, combinatorial mathematics
>
> explosion [ik'spləuʒən]
>
> n. 爆破，爆炸（声）；激增；爆发，迸发；突发的巨响；某事证实是错误的，推翻

        - There are many valid parse tree for a query
        - There are many possible ways to decorate each node
        - The total number of options grows exponentially (since most combinations of parse tree and decorations are valid)

> What are the total number of ways of doing the join? $A \bowtie B \bowtie C$
>
> $3! = 3 \times 2 \times 1$
>
> Each Join I have 4 different algorithms. $4 \times 4 \times 4 = 64$

    - Limited time
        - You do not want to spend 2 hours optimizing a query such that you save 2 seconds.

- You do not want to spend 2 hours optimizing a query such that you save 2 seconds.
- Information needed
  - A lot of information is needed for the optimizer to work properly
  - Recall the fundamental problem of query optimization?

> You don't know the best way to execute a query until you know the size of the Q result. But you don't know the result until you actually have to execute a query by that time. That is the fundamental problem that every database system has to solve one way or the other.

Measuring cost of a query

- Many factors contribute to time cost
  - *disk access, CPU*, and network *communication*
- Cost can be measured based on
  - **response time**, i.e. total elapsed time for answering query, or
  - total **resource consumption**
- We use total resource consumption as cost metric
  - Response time harder to estimate, and minimizing resource consumption is a good idea in a shared database
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
  - Network costs must be considered for parallel systems

> For now we go to ignore CPU time. We are we are assuming all the cost created are the time reading the data from the disk.
>
> in most database systems, they support certain kind of this.
>
> ```
> select f1(s,age)
> from student
> ```
>
> `f1`: is a user defined function
>
> Many data based systems support some version of this. Not all, but some

- We describe how estimate the cost of each operation
  - We do not include cost to writing output to disk

- Disk cost can be estimated as:
  - Number of seeks/rotates * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
- For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures
  - t_T – time to transfer one block
    - Assuming for simplicity that write cost is same as read cost
  - t_S – time to move from one block to a non consecutive block
    - If on the same track, then rotate
    - If on different track, then seek + rotate
  - Cost for b block transfers plus S seeks
    $b \times t_T + S \times t_S$

> I'm saying if you've once you find the right rotate location, reading from it and writing to it take the same amount, of course. Make your discussion easier. Make the formula shorter.
>
> But remember, we might have to write something in the middle. Remember merge sort. So because just because we are not writing the result of this doesn't mean we don't have to write anything to do this. Because remember, we are all. Most likely we are going to assume the data base too large. Your database outputs ten terabytes. You just cannot hope that you read the whole thing into memory. Now. Luckily, in real life, most of a database you deal with are small enough.

- t_S and t_T depend on where data is stored; with 4 KB blocks:
  - High end magnetic disk: t_S = 4 msec and t_T =0.1 msec
  - SSD: t_S = 20-90 microsec and t_T = 2-10 microsec for 4KB

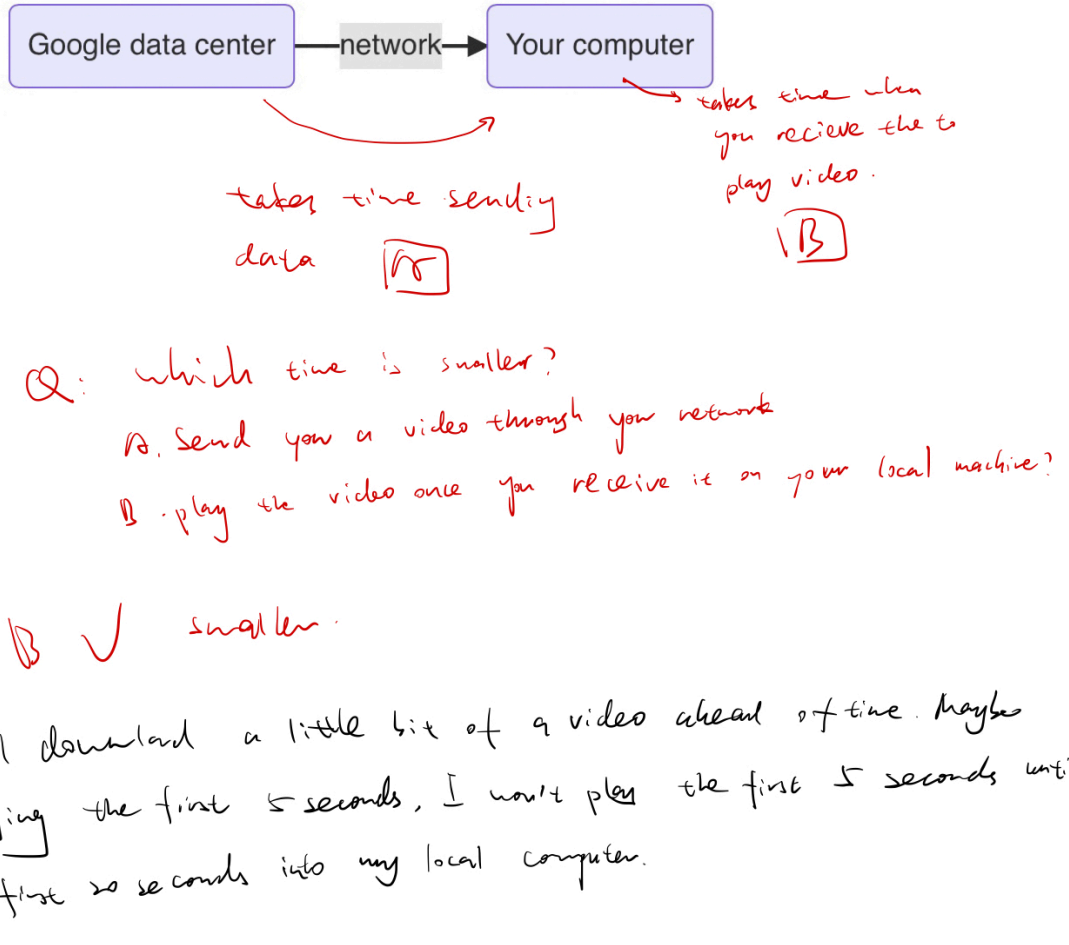
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation
- Several algorithms can reduce disk IO by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution

> What's buffer? You can store data temporaty into main memory. Typically, a database system will actually use some memory for you to store temporary data.
>
> For example

```
select *
from student
```

We will not assume, in the way to the students always in memory of enough.

Let's say let's use YouTube as example.



For example:

Maybe a previous query also asked about the student table. That previous query the database, decide to actually load the student table and if no significant part of student table in the main memory. So next time you asked another used to ask the query on the student table. It doesn't have to go to the disk to fetch the whole table. Typically we don't take that into account even between queries. So if we want to do something we do not know and we do not assume that one will leave something in the buffer. Now a smart query system will take that into account. A smart grid optimization system will take that into account. For the sake of this class, we don't. Once again, we want to make caculation simpler.

And however, later on, you will see a something like this.

```
select
from A,B
```

If I have two tables, we are doing a join. And within that, we in this query, we may make use of buffer to

> speed things up. So I may actually read table A into some buffer and then do the calculations that we will actually keep into account. We will actually put that in now. So we do not concern buffer management between different queries

- Worst case estimates assume that no data is initially in buffer and only the minimum amount of memory needed for the operation is available
    - But more optimistic estimates are used in practice

Query processing for individual operations

- Selection - σ
- Joins - $\bowtie$
- Projection - $\prod$
- Ordering - `ODEAR BY`
- Group By - `Group By`

> So before we can talk about optimization, we need to first look at how would a database system execute its operations.

15.3 Selection

- $\sigma_{scondition}(Table)$

    > when I say I do selection, I do selection on a single table. If you want if you want to leave multiple people, we do a join. So selection is always on a single table and then it's always based on a condition.

- A few considerations
    - Condition
        - Can be “attribute = value” (equality)

            > The simplest type is what we call a quality condition.

        - Or “attribute <= value” (or <, >, >=) (range/comparison)

            > This has to be something that you can order, right? Like integers through numbers, string, even string. You can order it, but typically alphabetical order. This is what we call typically a compression career, and more often we call it range query.

        - Single condition or multiple condition
            - E.g. $(x = 1\ AND\ y = 3), (x = 3\ OR\ y = 4)$

> $x > y$
>
> This will be super interesting.
>
> ```
> select gpa > age/5 from (Student)
> ```
>
> It's a trick question to ask you to reveal your age. If a guy asked you this query, make sure you don't answer this query.

- Attribute
  - Primary key or not

> So the attribute that you are asking, whether it's a primary key or whether it is part of a primary key, whether it is unique or not, these things also affect it.
>
> Q: Why do you think knowing that this actually is unique is can be important?
>
> $\sigma_{a=1}(Student)$
>
> If I know execution is unique, what can I immediately see? How many tuples will this query return? It will return 1 or 0; But remember, always consider possibly that you don't return anything. Then you have to ask, do we have an index? If we have an index on attributes that they have and what kind of? Is it B^+ tree? Is it a hash table?
>
> Q: Why does why is it important to check whether an index is the best really hash table?
>
> $\sigma_{age>25}(Student)$ Do you think a hash table on `age` will help in this query.
>
> A: No, It's hash table, no requirement for ordering. You just have the same value being hash, the same bucket. But that's no guarantee. Like, one, two, three, four, five will be in order. So if you see a query and you say, hey, that's a hash table, there has to be is probably not going to be helpful.
>
> But B^+tree can still be very useful. You'll find the first page. If it is part of the index, you just scan through. Go through the leaves. Because B^+-tree you have the links between consecutive.
>
> But on the other hand, if this is the case $\sigma_{age=25}(Student)$, then certainly you should at least consider a hash table. Because now all I need is all tuples put aside by one condition.

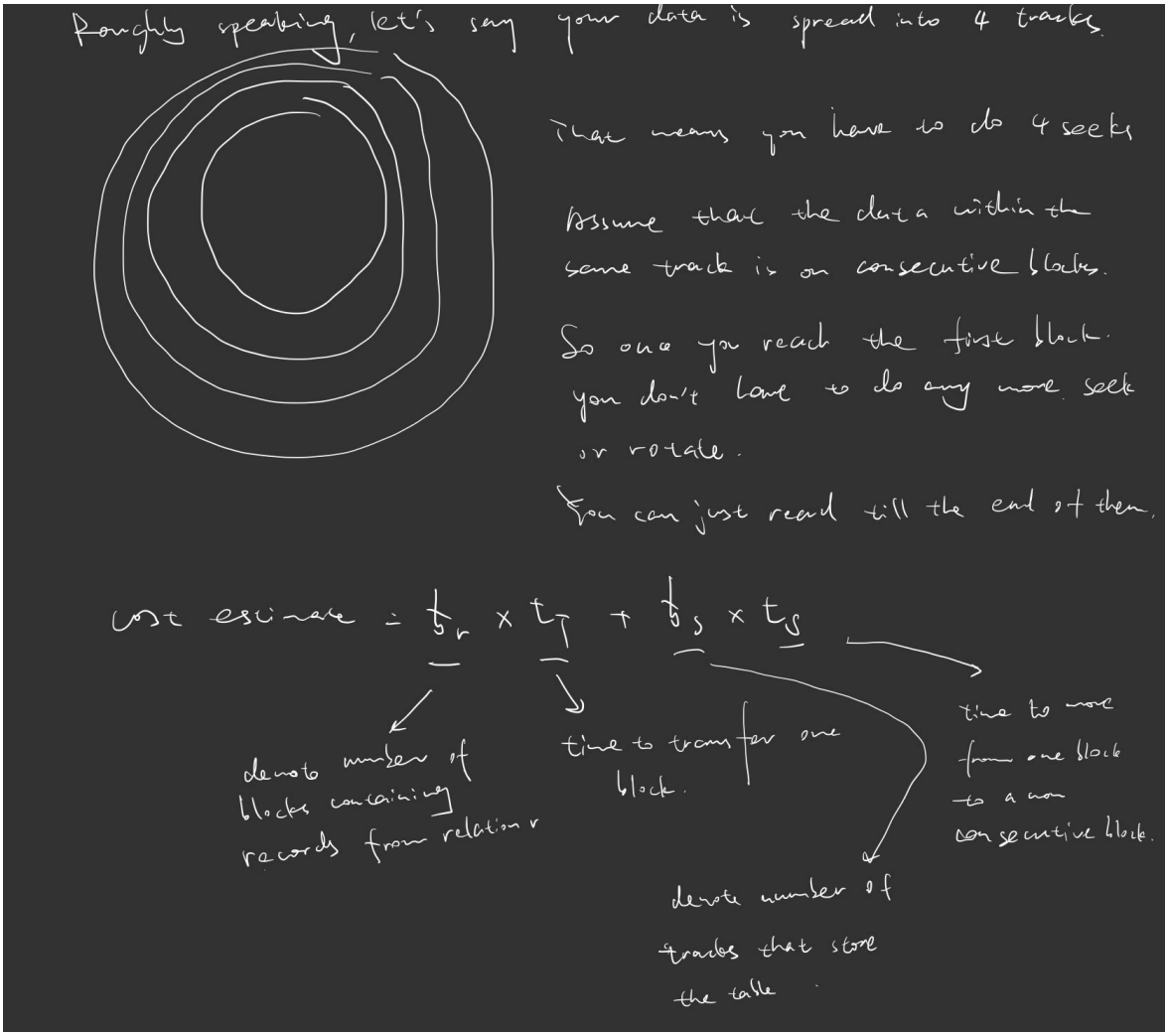
- Organization
  - Index available?
  - What kind of index?

- Basic case: File scan / sequential scan

> Q: What's the basic? What's brute force method?
>
> A: you just read the whole table from beginning to end and just examine every to potentially very satisfied position.
>
> Q: So why? why do we conside this? Now, obviously, this work, this will give you the right answer. This does sound stupid? But why do we even think about it?
>
> A: Because sometimes there's no other way, right? If you have no index, no whatever. That's the only thing you can do. Read the whole tabletop beginning to end. And there are some cases where even you have indexes or whatever, you don't want to use it. We call this typically we use the term `sequential scan` to describe this. No trick. No optimization, just we're forced to thing.
>
> Good thing: you don't need to care about the condition. It works.

- Just read the whole file block-by-block from beginning to end and check if each tuple satisfy conditions
  - Cost estimate = $b_r * t_T + b_s * t_S$
    - b_r denotes number of blocks containing records from relation r
    - b_s denotes number of tracks that store the table

      > t_T – time to transfer one block
      >
      > t_S – time to move from one block to a non consecutive block



- If selection is on a key attribute, can stop on finding record
  - Average cost = $\frac{b_r}{2} * t_T + ? * t_S *$
  - ? Is harder to predict, is roughly max$(1, \frac{b_s}{2})$

```
select ssn= '12345678' from Student
```

Let's say ssn is your primary key. You keep reading, and once you find the tuples you can stop. Because it is a key, that means 1 or 0 tuples. But now you find the one->stop.

So that means now the actual cost may differ depending on whether you are lucky or not.

And it is whether you are lucky or not. We usually call the average. So avareage is the half (1+0)/2

But how many tracks you want to read can be a bit tough to estimate because you don't know which you are going first and so on. So that can be a bit tricky. Typically, we assume we do have a track. Therefore, 1 is limit.

- Linear search can be applied regardless of
  - selection condition or

- ordering of records in the file, or
- availability of indices

> This linear search says sequential scan can be executed on any condition, any attribute. So every database engine, when they try to estimate what is the best way to do a query, they always calculate this cost as a baseline. So if I want to do something else. I need to make sure that that method beat these methods. If not, you know what? Let's not do anything. Fancy that. Just read the whole table. Boring. But it works.

- Note: binary search generally does not make even if data is sorted except when there is an index available, as each step will require a rotation

> You start searching in the middle and then you jump to halfway. That jump to half way is a rotation. Remember, that is actually quite costly, So even though your table is sorted on the attribute, binary search it's not often use. Unless your track have a very large capacity.
>
> So that makes justifying a binary search much more complicated. So most system don't even consider that even if your data is maybe you have a B^+ tree, A clustering B^+ tree. Even that your data is spread into multiple this, hat it is a risk because when you jump from the middle to the next stop, it may be on a different track. Then you have to do a seek plus a rotate. And when you jump from the second half to the next half, it might be another track. Then that's another seek. So that's where I hope that you start to see the difference between accessing data from database versus accessing data directly from public memory. Because it's not just the problem of complexity.
>
> Obvisouly binary search is logn, sequential search is n, but that logn can be very large

```
select *
from student
```

the index will not helpful

```
select *
from student
where age > 25
```

If you have a clustering index on age will be helpful. If you have a non cluster index on a then if age is not a primary key, then probably no help. Unless the student is a student of kindergarten.

For example, if your database system to say hey what is the range of value? The range of age is between three and five. So that's where DBA earned their money. By able to working out the query and having the database. Obviously you can not tell database every time you ask the query of the terror database beforehand. Obviously you can not tell database every time you ask the query of the terror database beforehand. If you tell the database too many things, your database need a lot of space to store that thing and a lot of time could retrieve the things that query. So that's another debate. There's no free lunch. There's absolutely no free lunch here. But you also need to know that there are things that the database

system are supposed to do.

- If you have an index: Index scan
  - Use an index to search.
  - Attribute of index needs to match condition
  - Also hash table is not useful for range queries

    ```
    select age>=25
    from Student
    ```

    If you have a index on GPA, that index is helpless. So you have to match the attribute. Also, as we talk about hash table, it's not useful for each query we talk about that to.

- Assume that the query return t tuples, stored in b_r blocks (notice that $t > b_r$ – often by at least one order of magnitude)
  - $t = 1$ if equality search on an unique attribute

    because there's only one to that match.

- Clustering index:
  - $Cost = time\ for\ searching\ the\ index + b_r \times t_T + \alpha \times t_S$
    - α is the number of tracks that contains retrieved tuples (if $t = 1, \alpha = 1$)
    - α grow slowly (if at all) with t (since data is clustered)

    b_r denotes number of blocks containing records from relation r

    b_s denotes number of tracks that store the table

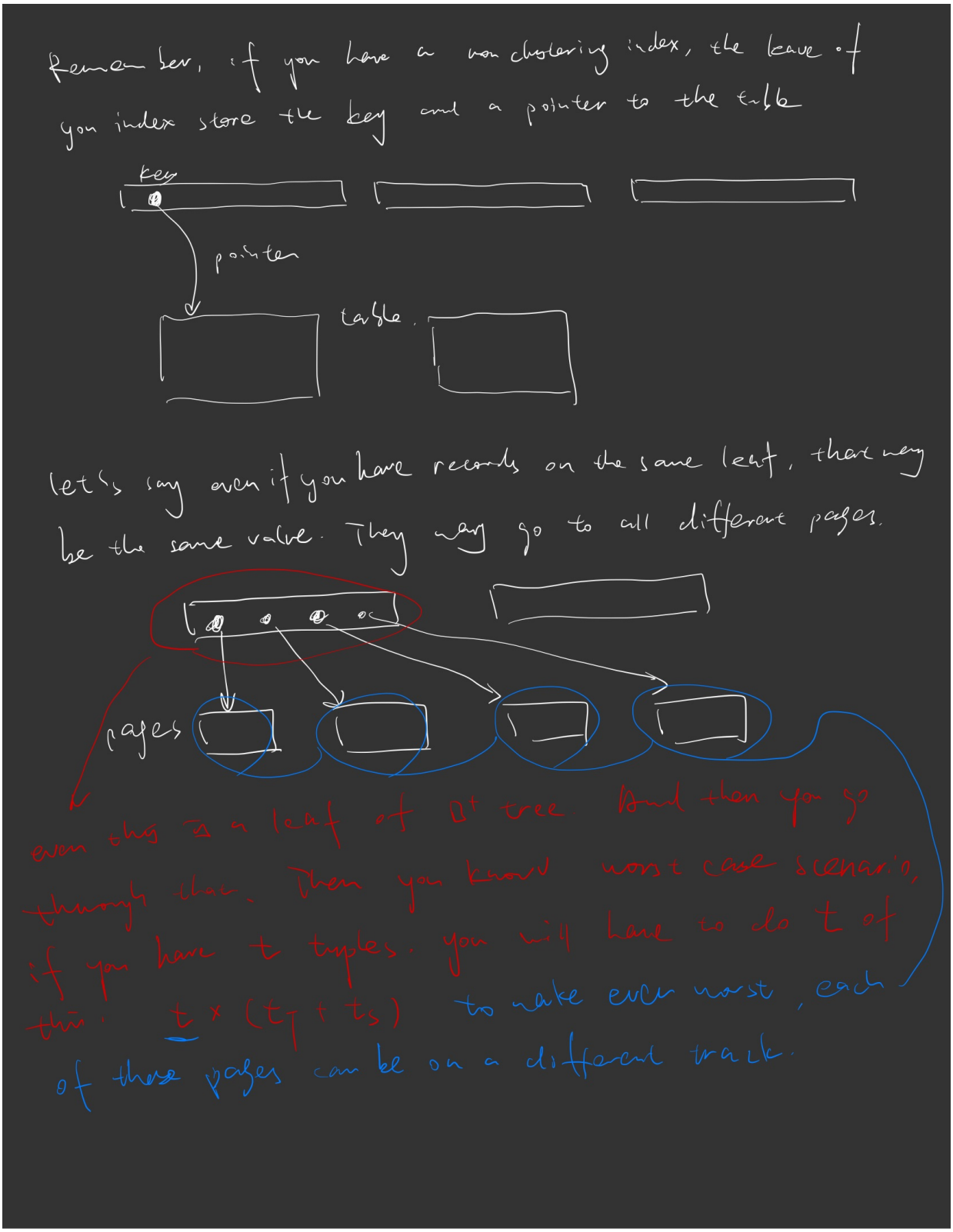
    t_T – time to transfer one block

    t_S – time to move from one block to a non consecutive block

  that means what the data is store. So in the order or at the very least everything that has the same value is on the same page. Then you can actually calculate based on b_r . Based on number of blocks. Right now you have to worry about obviously, the number of tracks. The good news is that your data is really cluster nicely, then it is far less likely you have to seek multiple tracks unless you're clearly told a lot of things.

- If you have an index: Index scan
  - Use an index to search.
  - Attribute of index needs to match condition
- Assume that the query return t tuples, stored in b_r blocks (notice that $t > b_r$ – often by at least one order of magnitude)
  - t = 1 if equality search on an unique attribute

- Non-Clustering index:
  - $Worst\ case\ Cost = \ time\ for\ searching\ the\ index + t \times (t_T + t_S)$



    - It may be faster if the DBMS read all the index record and determine which tracks the data are

stored before fetching the record

- Even this can be infeasible
  - For example, if there is an ORDER BY clause and there are too many records being selected to fit in main memory
    - Cannot bring them all in to sort

> But in virtually all database system, if you define primary key, that data will be organized at least based on the by primary key. You have a clustering, B plus three on the on the primary key, or you actually start the table based on the primary key, whatever that means. If you ask an extra bill on a secondary index, for example, your primary key SSN and then you are backing, that's how you carry over the GPA. You are backing the GPA is ordered the same as your SSN.
>
> The number of tuples return become of primary importance. If this could return to two tuples, I'm betting you'll fine. It return 2000 tuples, I think you should forget index.

- Conjunction: $\sigma_{\theta_1 \wedge_{\theta_2} \wedge \cdots \wedge_{\theta_n}}(r)$
  - Now multiple indices may be available
  - Option 1: Single index
    - Use one of the available index
    - Read in tuples into main memory, then apply other conditions
    - Some cost estimation as before, only with t (and b_r) are tuples/blocks that contain tuples that satisfy the condition that the index is correspond to
  - Option 2: Clustering multiple-attribute index (if available)

    > There is also another option or what we call a multi attribute index, which probably is a good time to interpret any question the previous conflicts. So we talk about having an index on GPA, having an index on SSN and have an index on age. There's nothing stopping up to having an index on a combination of all. In the key restrictions that is still, let's say you want to use a, B plus three, you still need to be able to order the thing. So, for instance, I can actually create index on the combination of age and GPA.
    >
    > So for each tuple, I'm going to create index record on age and GPA.
    >
    > [(age, gpa)] Now the question becomes, how do we order them? Oder by age first. For example, (19, 2.1) , (19, 2.2), (19. 3.0), (20, 2.7), (20, 3.4) so on. Most database system now they allow you to create index like this. What do you think are the pros and cons of such kind of index? Once again, no free lunch. It obviously is a bit more expensive because obviously each index record is now having a larger number of bits. Notice that even though if I have this. If I want to ask the query that say select start from student where GPA is greater than 3.5.
    >
    > ```
    > select * student from
    > where gpa >= 3.5
    > ```
    >
    > you cannot use this index. Because the records are not sorted by GPA. It is sorted by a first and

> only if they have the same age. So it is actually quite limited.

- Option 3: Intersection of identifiers
  - Consider all secondary indices that associate with an attribute in the condition
  - Query the index, read in all index records into main memory (do not go to database yet)
  - Only select pages that are in the answers for all indices
  - Read the tuples and subsequently apply other conditions

```
select *
from student
where age >= 19 AND
       gpa >= 3.5
```

Notice that you have two conditions, right, in this case. You have a condition on age. You have a condition on gpa. Now assume the primary keys to their I.D. So the clustering index is going to be on student ID, any other index fund cluster.

Now you have one index on age. Now you have two options. You either do use the index or don't use.

Students might from smu, high school. Therefore it depends.

What if I have both index? I have a secondary index of age as well as a secondary index of GPA. What are my options?

```
age >= 19 AND gpa >=3.5
```

What does and mean? That means I have for any tuples to be selective, it has to satisfy both conditions. So let's say we have 10,000 students age>= 19. And 50,000 students gpa at least 3.5. How many student will satisfy the condition? => less than 50,000. 10,000 at most. Or it can be very small.

```
aga >= 19 AND gpa >= 3.975
```

age>= 19, we have 10,000 students; gpa >= 3.975 we have 5,000 students

Now the question NO.1is that you have to guess how small thing you have. How do you guess whether this is actually small? We'll talk about that later on. But let's assume if this is more and you think this is small enough, what is my fault option? Now all I have is non-clustering index on both table both attributes. I don't have a clusering index for this attribute.

What is my fault option?

The point here is that you still have to get these tuples, you might have to get a lot of tuples before you

can actually in the set or regret maybe even the whole table. But, there is a bug here. I do not have to retrieve tuple immediately after I looked at the first index. So I look at the first index, let's say the first index will tell me what where each tuple says, find these conditions. So you can least say <page 1, page 2, page 10, page 12, page 13 .... > so on so forth. I do not go to those page immediately. I go to now look at that index from the GPA. That index also tell me where are the tuple? Just satisfy this condition `gpa >=3.975` <page 4, page 10, page 14 ... > If these are the list, what do I mean? I don't need to go to page 1. I don't need to go to page 4, page 12, page 13, page 14. There is at least some hope it will cut down on the number of pages to read. Notice that page 10 in the both, doesn't mean you have a duplicate and satisfied condition. Because they may be two separate tuples. But it also may be turn out the same too.

But the idea is that there is at least hope that you can cut down on the number of pages. And you have some confidence that this condition actually returned very few tuples, even though in the week your conditions return a lot of tuples. Remember, you don't access tuples until the very end, you don't go to the main table until you figure out what other pages. So if you believe that this condition where you feel tuples satisfy both conditions simultaneously, this is a risk that a database system might want to take. Instead of just forget about it and go without it.

It is not that rare that in a real application you will after query like select start from table A

```
select *
from A
where condition 1
and condition 2
and condition 3
...
```

So there might be a lot of condition. It happens less read than you think. If that's the case. What you hope is that the career optimizer who at least be smart enough to consider that option.

However, there is one situation where this index is a deal.

```
select A.gpa, A.age
from A
where A.age>= 19
      And A.gpa >= 3.567
```

I will actually use this index. Why? I don't care how many tuples I'm going to be, I'm still going to return, I'm still going to use datas. So I actually have an index that I talk about earlier. Do I even need to bother to go to the main table? I'm not going to go to the table. Right. Even if we actually keep it for everything that I need.

So your database system need to figure out, hey, for this query, I'm going to use an index, I don't care about how many people are going to return because I don't even need to touch the main table.

So that means what? That means, if you have application, if you feel like these kind of queries like this

query are going to be asked a lot, then there might be a reason for you to actually create that index, even if they eat up quite a bit of space. Now, of course, there's still no free lunch. Because why? Because your table is being updated often. Then the cost update will be. So always no free lunch. But you can get a cheap lunch.

$\sigma_{\theta_1 \wedge_{\theta_2} \wedge \cdots \wedge_{\theta_n}}(r)$: or

```
select *
from student
where age >= 20
   OR gpa >= 3.5
```

If I have an index of age, let's just say that's the only index I have. Should I use it? That does not seem attractive.

So if you look at a single index on a single attribute, a bunch of all query, that single index is practically useless. Now, on the other hand, you still can kind of pull district a little bit. Because what you can do is that you can once again look at the index for all one quotation. Now if a page in any one of the index. I said, read those take. But if each of them is small enough, maybe the total result is still going to be small enough that make it worth. So you can still kind of pull this trick. Is it going to be effective? Probably not very likely, but at least something if you got this query.

```
select *
from student        // from dallas middle school
where age >= 20
   OR gpa >= 3.5
```

Each of these conditions returns very few to tuples, very few pages. Even if you combine them together. Remember, in this case, we combine we have to take every page that of here. See the point here? So but you have to be once again very sure that each of these query actually returned a very few number of tuples. But the point is you have it all query. If you just have a single index, forget.

- Disjunction:$\sigma_{\theta_1 \vee_{\theta_2} \vee \cdots \vee_{\theta_n}}(r)$
  - Now multiple indices may be avaiable
- Option 1: Intersection of identifiers
  - Assume indices present for ALL attributes in the condition
    - Query the indices, read in all index records into main memory (do not go to database yet)
    - Only select pages that are in at least on of the indices
    - Usually is expensive

        - Notice that it doesn't work if indices for some attributes in the condition is not available
    - Option 2: Sequential scan, apply condition when tuple is read

> In fact, if you do not have an index for every condition you should forget. And just do a sequential step. So kind of a query, is that all query is very hot to optimize in the database system. If you ask a query, that is all conditions, you probably should not expect the database to execute fast. You probably should expect they just to a sequential scan it.

> Now, once again, semantics over semantics and database design go first. Don't cut corners just because it's less efficient, very dangerous. But you at least have a sense or query. There's really nothing much to criticize I can do.

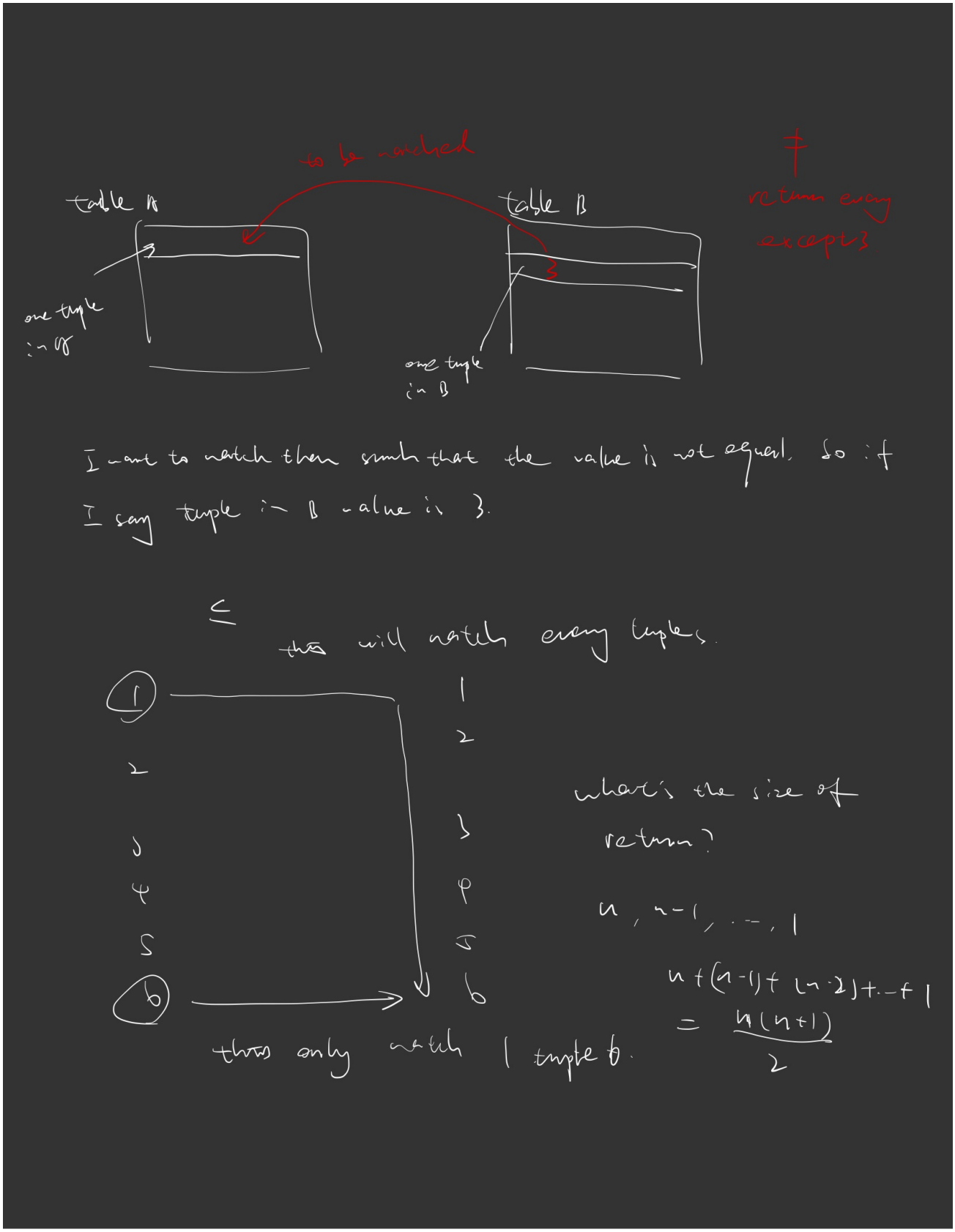
15.5 Joins

> What is the worst case scenario if you do a join in terms of efficiency across. Every pair matches. Then you have an tuple in the first table and an tuple in the second table you are returing and squred tuples. So in the worst case, it's going to be contracted operations. Unless you do some of the techniques, at least blindly speaking, at least if you don't, you don't play any tricks. But the reality is the number of pupils we've touched is actually, I don't get how efficient you are in the midst of this.
>
> People spend a lot of time optimizing this query.

- $R \bowtie_{cond} S$

  > Now this condition is really important.
  >
  > Query 1:
  >
  > ```
  > select *
  > from A, B
  > where A.a = B.b
  > ```
  >
  > Query 2:
  >
  > ```
  > select *
  > from A, B
  > where A.c <= B.d
  > ```
  >
  > Query 3:
  >
  > ```
  > select *
  > from A, B
  > where A.e != B.f
  > ```
  >
  > Just a blind guess which of these three queries returned the most number of troops?

Query 3



- Most common case: condition is an equality condition between attributes of R and S
  - We call this equi-join

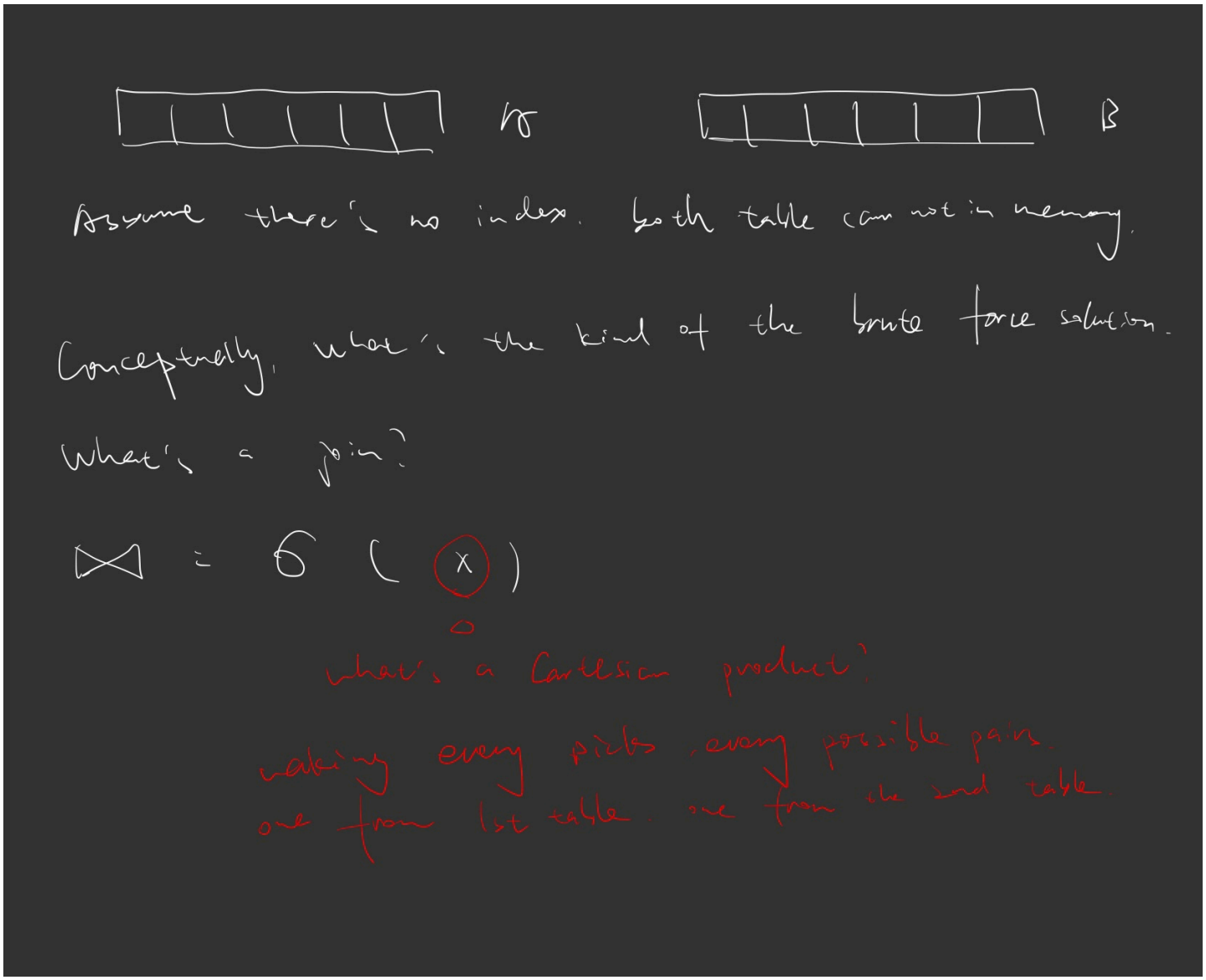
```
select *
from A,B
where A.a = B.b
```

`=` this is the equi condition. We are going to focus our discussion on this mostly, probably nearly exclusively. I will probably make a passing remark. It is less than or equal to and it is not equal then. There are a lot of ways for you to execute this query in this non-equity join then really that you only have really few options now.

So give me a can you think of a algorithm that will work no matter what the condition is?

- Example: linking an attribute with its foreign key

- With equi-join there are a lot of options
- With non equi-join there are very few



> t_r and t_s are tuples;
>
> $t_r \cdot t_s$: denotes the tuple constructed by concatenating the attribute values of tuples t_r and t_s
>
> n_r: the number of tuples in r
>
> n_s: the number of tuples in s
>
> $n_r * n_s$: the number of pairs of tuples.

15.5.1 Joins – Nested loop

- The naïve algorithm
- $R \bowtie_{cond} S$

```
for each tuple tr in R do begin
  for each tuple ts in S do begin
    test pair (tr,ts) to see if they satisyf the join condition cond
    if they do, add tr * ts to the result                /// ?????
  end
end
```

> I'm not calling this smart. But I'm calling this always will. Notice that this work, no matter what the condition is. This is naive, but it will work.

- R is called the outer relation (outer loop), S is the inner relation (inner loop)
  - Either relation can be in the outer loop
  - Flipping R and S will give you the same results

> ```
> for each tuple ts in S do begin
>   for each tuple tr in R do begin
>     test pair (tr,ts) to see if they satisyf the join condition cond
>     if they do, add tr * ts to the result
>   end
> end
> ```

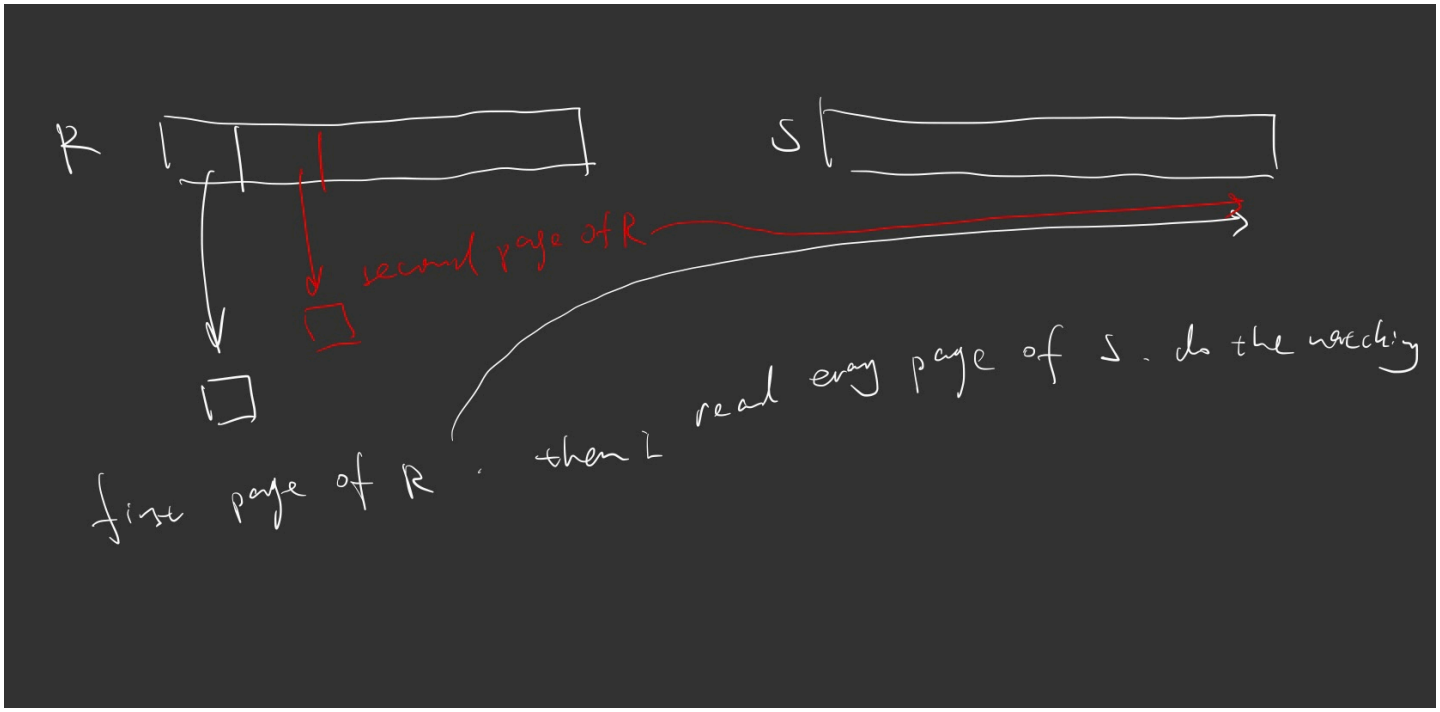
- Works for any condition

> What if both tables are too big for many members. Now you have some memory of it. Remember, in order for me to process things. I have to make sure both tuples. Have to be minimum before we can process. But why does it matter? Well, we have to actually look deeper in.

15.5.2 Blocked Nested loop

- Modify the algorithm for secondary storage

```
for each block br in R do begin
  Read br from disk into main memory
  for each block bs in S do begin
    Read bs from disk into main memory
    do a nested loop for each pair(tr,ts)[tr ∈ br, ts ∈ bs) to see if they satisfy the
join condition cond
    if they do, add tr • ts to the result.
  end
end
```



- Running time depends on amount of main memory buffers available
  - Need at least 2.1 for R and 1 for S
- Case 1: Buffer large enough to hold all blocks for both tables
  - Read both tables into main memory and then loop all the tuples inside
  - Cost = cost for sequential scan for R + cost for sequential scan for S

> if the two tables are actually small enough such that both men memory, well, the solution is simple. Just everything in memory and just let the CPU do its thing.
>
> Cost for doing the sequential scan plus the cost of doing this scan of this.

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 2: Minimum number of buffers (ctd)
  - Separate cost into page access and seeks
  - For page access, each block in the outer loop need to be read once
  - For inner loop, each block has to be read once for each block of the outer loop
  - Number of block access = $b_r + b_r \times b_s$

    > b_r : R has b_r pages
    >
    > b_s : S has b_s pages

  - Question: given R and S, which table should be in the outer loop

> Smaller table should be outer loop.

> Once again, these are things that as a user, don't worry about this, but as database system means the who want to build the next version of database system them met.

- Running time depends on amount of main memory buffers available
  - Need at least 2.1 for R and 1 for S
- Case 2: Minimum number of buffers
  - For seeks/rotate
  - Each page in the outerloop need to be seeked
  - Assume query is not being interrupted, the inner loop is being read in consecutively
    - So only the minimum number of seeks
  - Number of seeks = b_r* (number of seeks to read S)
    - However, it is more than likely that there will be interruptions
    - For example: the disk head may have moved pass while the joining is in process
    - Worst case scenario: $b_r * b_s$

      > Then you would the total cost would be $b_r^2 + b_s^2$

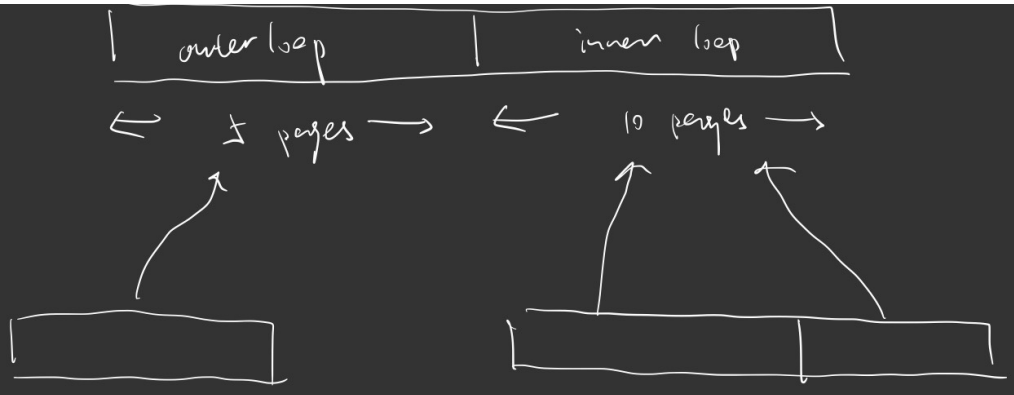
  - Which table should be in the outerloop?

> So now we look at two extremes, right? One extremely is that I have so much memory. I read the two tables in my memory. Who case? Second extreme, he said, I really have no memory. There's only one package for each table. I better put the smaller table in the outer loop. So let's take some middle of the road case. So there are, let's say, a few blocks. It's not big but enough to read either of the table. So in addition, a weak table is in outer loop. I have one more thing to worry about. How many pages devoted to
>
> the outer loop, how many pages do I need to be able to inner loop? Now my algorithms like this.

- Case 3: Enough buffer to fit either R and S (plus k buffers for the other table), but not both
  - (Assume the buffer fits table S)
  - Read S into main memory
  - Then read R into main memory (in steps, because not enough memory to read it all at once)
  - Join pairs of tuples in R and S
  - Cost = cost of reading S (sequential scan) + cost of reading R
    - Depends on whether the query get interrupted, it can be as little as the same as sequential scan or as much as ceiling($\frac{b_r}{k}$), where k is the number of buffers allocated to R

- Running time depends on amount of main memory buffers available
  - Need at least 2.1 for R and 1 for S
- Case 4: None of the above (k buffers available between the two tables)
  - We will need to assign buffers to each table
  - Assume we assume k' buffers to R
    - Then k-k' buffers to S

```
for each k' blocks in R do begin
  Read k' blocks of R from disk into main memory
  for each (k-k') block bs in s do begin
    Read (k-k') blocks of S from disk into mian memory
    do a nested loop for each pair (tR ts) [tr ∈ br, ts ∈ bs] to see if they
satisfy the join condition cond
    if they do, add tr • ts to the result
```



I will first read the first five pages of loop into the buffer.

Then I will repeat the inner loop. Ten page at a time. Until the whole thing is finished.

I joined a tuple and then I read the next five pages on the outer loop, put it there, and the I read everything here.

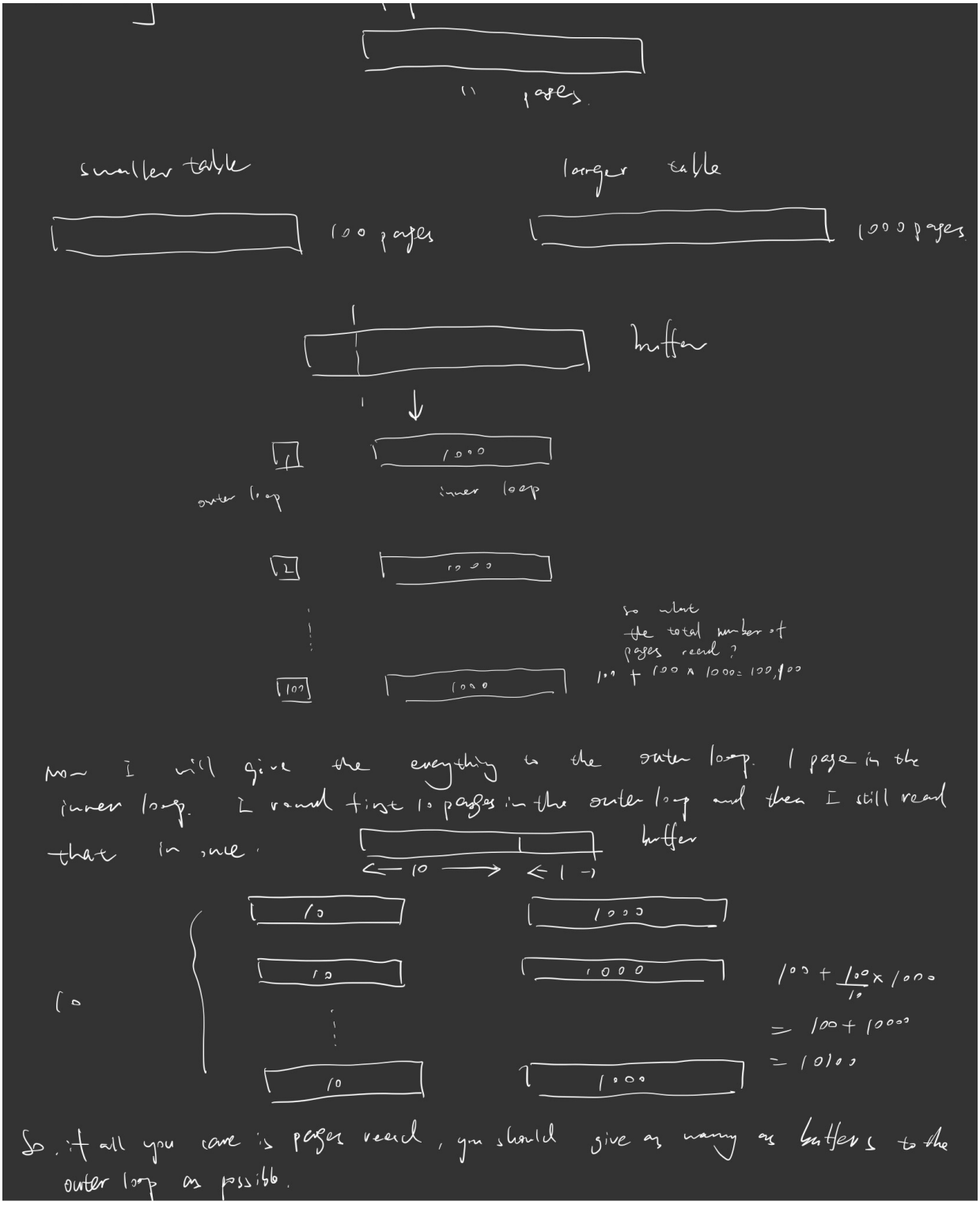
Question: If I have 12 buffers, how many buffer should I keep the outer loop, how many buffers should I keep in the inner loop. Assume we ignore time and rotation all time. I only care about number of pages. Obviously I have to get one page to each of outer loop and inner loop at least.

- Running time depends on amount of main memory buffers available
  - Need at least 2. 1 for R and 1 for S
- Case 4: None of the above (k buffers available between the two tables)
  - Cost: Consider number of blocks read
    - Outer loop: every block in R need to be read once – cost = b_r

      > the outer loop is read once

    - Inner loop: for each iteration of the outer loop, the whole table in the inner loop need to be read
    - The outer loop executed ceiling($\frac{b_r}{k'}$) times
    - So the total number of block reads for inner loop = $\frac{b_r}{k'} * b_s$
  - Given that, what should be the value of k'?

    > because you have k' blocks for inner loop.



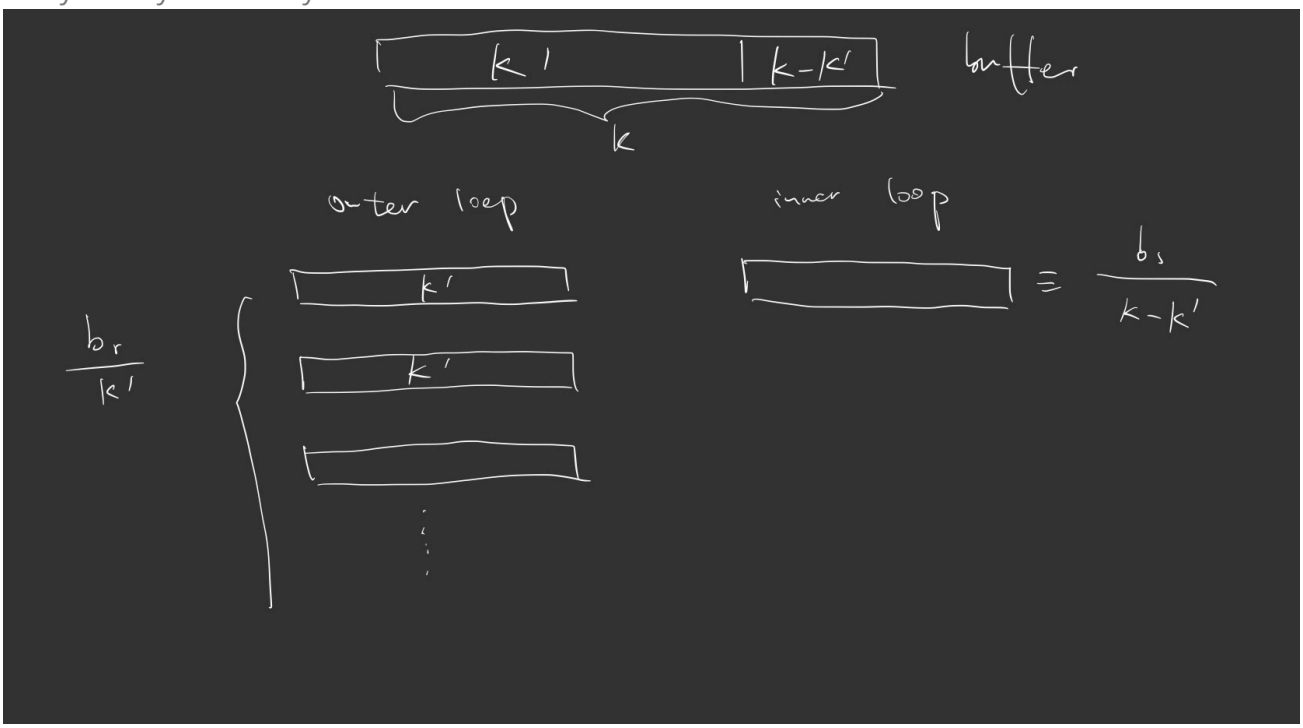
Number of 4 acutally become a big trick here.

- Running time depends on amount of main memory buffers available
  - Need at least 2.1 for R and 1 for S
- Case 4: None of the above (k buffers available between the two tables)
  - Cost: Consider number of seeks

- Cost: Consider number of seeks
  - Outer loop: every time the outerloop executes there need to be a seek (why? – similar to case 2)
    - Number of seeks = ceiling($\frac{b_r}{k'}$)

      > Each iteration, there will be one seek
  - Inner loop: Similar to case 2
    - Best case scenario = $b_r * (number\ of\ seeks\ for\ sequential\ read\ S)/k' *$
    - Worst case = $\frac{b_r}{k'} * \frac{b_s}{k-k'}$
- Now what should the value of k' be?

> Now, this is a very pessimistic estimate because we assume each time we read a segment of the inner loop that's going to be a seek and it may or may not be the case. So this is a very pessimistic calculation. so once again, if you want to be optimistic this way, you might be smaller than this is `k' * (k-k')` But then once again, then it depends on whether your system want to be willing to be pessimistic or not. But these are some initial calculations.
>
> I may or may not allow you to do differentiation in the final.



- Should we use an index?
- Index for outer loop is useless
  - Unless clustering index on join attribute where you can read the table based on the join attribute, and that attribute is not unique (why?) > always never happens

> Because you have to root for every atom anyway. You have to go through every tuple in the outer loop. So index ultimately is pretty much useless

- How about inner loop
  - To use an index, that means for each **tuple** there need to be a search
    - Number of tuples is larger than number of blocks
    - If each tuples only match with very few tuples, it's fine
    - However, if there are potential large number of matches with a secondary index, things can get dicey.... (exercise)

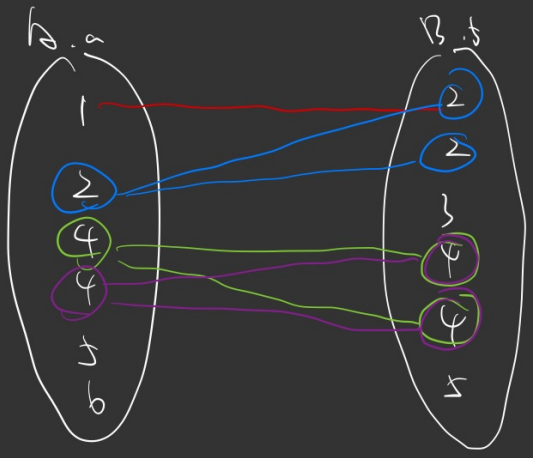
> ```
> select *
> from A,B
> where A.a = B.b
> ```
>
> If we decide to do nested loop on A, B and A is outer and B is inner loop

> We have clustering, that means data that have the same value is going to be the same page. But when we match on values.
>
> Let's say, we have 1,000 tuples and there duplicate on the attribute how many values are there, must be less than 1,000 , right? 1,000 tuples can only have a thousand variables in that. I would not distinguish that. At the very least you have potential to join less than.

????

Assume the outer loop is sorted, inner loop is also sorted.

```
select  *
from    R, B
where   R.a = B.b
```



How do I join that to make it efficient?

Let's compare the first pair.

(1, 2)

They don't match. therefore (1, 2) these tuples can be ignore

Why?

Because 1 is already the smallest one in the B.b and it's already larger than your value.

Let's say the two tables are not sorted on these two

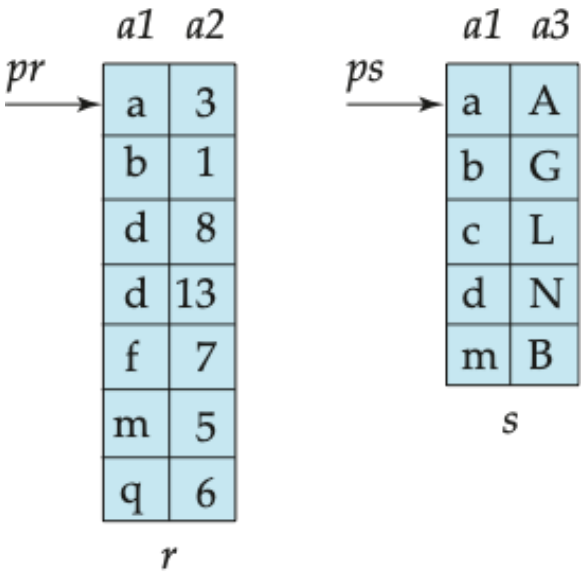
You can't do that. You have to sort first.

15.5.4 Joins – Sort-merge

- Consider an equi-join: $R \bowtie_{R.a=S.b} S$

> Notice that this parctically only work with equi-join: =

- Assume
  - R is a sequential file ordered by attribute a
  - S is s sequential file ordered by attribute b
- Now to join the two tables
  - We can use the merge algorithm from merge sort



> If there is 100 page of table R with d, and 2,00 page of table S with d, then you essentially do nested loop for those pages. So you can not guarantee linear. So it will be perfect if there's no duplication. Let's say you're joining primary keys, for instance. Both table is primary key. But even if only one table is primary key, there's still a danger that you have to go back and forth a little bit.

> If there's one table in the data that is unique, you probably want to put the non unique table in the outer loop. So each time you go inner loop, you're moving the outer loop, but not the inner loop.

- Key difference
  - If attributes have duplicate values then one may have to go “back and forth”

> The only difference from merge sort is this.

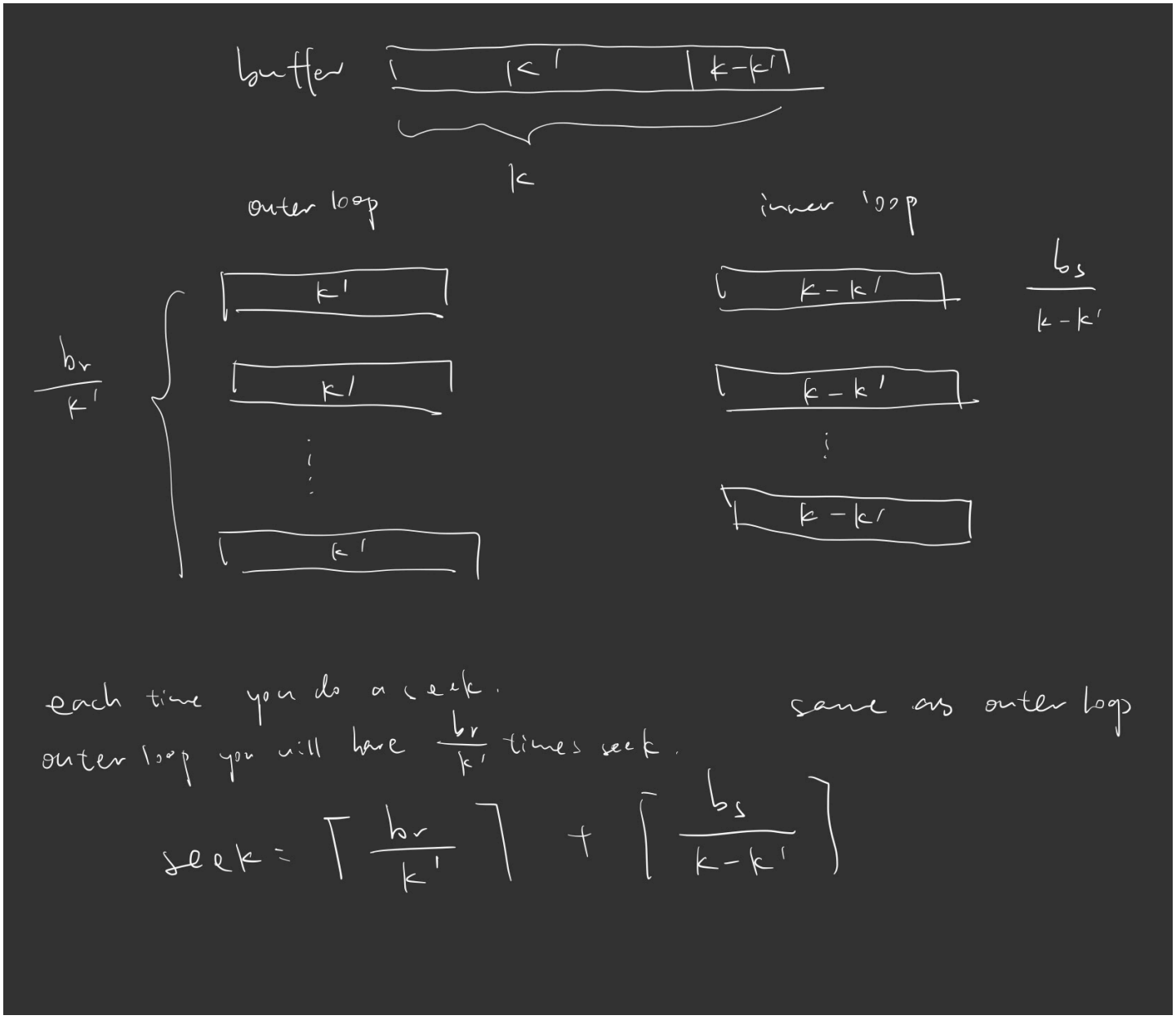
- Cost for the merge
  - Assume we have k' buffers for R and $k–k'$ buffers for S
  - Blocks read = $b_r + b_s$ (best case, or unique value of attributes)

> In the best case, every page only need to be read once.

- In general, may multiply by some factor m to take account for duplicates

- Seeks = $ceiling(b_r/k') + ceiling(b_s/(k–k'))$ (why?)

> And the seats are basically nothing special



- What if the tables are not sorted?

- What if the tables are not sorted?

> If the table is not sorted and you still want to do it, there's only one way to go. -> sorted.

- Sort them (!) and store the sorted table temporarily
- Then apply the merge algorithm
- This is known as the sort-merge algorithm

> You sort firs then you merge . Not merge sort
>
> Sort-mege is an algorithm that you join two tables. Merge sort is an sorting algrorithm.
>
> So how do we start sort? -> use merge-sort

- Cost = cost to sort the tables + merge cost (as previous slide)

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
- Assume we have 11 pages of buffers
- Consider sorting R
  - First step: Divide page into segments of 10 pages = 100 segments (to make analysis easier)

    > You can do 11, I just want the number look nice.

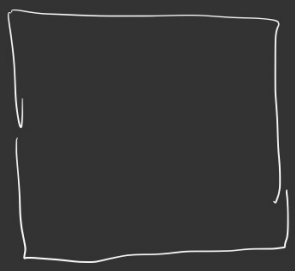
  - Merging step: assume merge 10 segment as a time
    - Two iterations: 100 segments, 10 page each -> 10 segments, 100 page each -> sorted, 1000 pages
  - Total number of read/writes = 3 (total iterations) * 2 (read/write) * 1000 = 60000

> The first iteration to create this then two steps to merge. So total time is three times two because you have to do and write a thousand.

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
- Assume we have 11 pages of buffers
- Consider sorting S
  - First step: Divide page into segments of 10 pages = 10000 segments (similar as R)
  - Merging step: assume merge 10 segment as a time
    - Four iterations : 10000 segments, 10 page each -> 1000 segments, 100 page each -> 100 segments, 1000 page each -> 10 segments, 10000 pages -> sorted, 100000 pages

- Total number of read/writes = 5 (total iterations) * 2 (read/write) * 100000 = 1000000

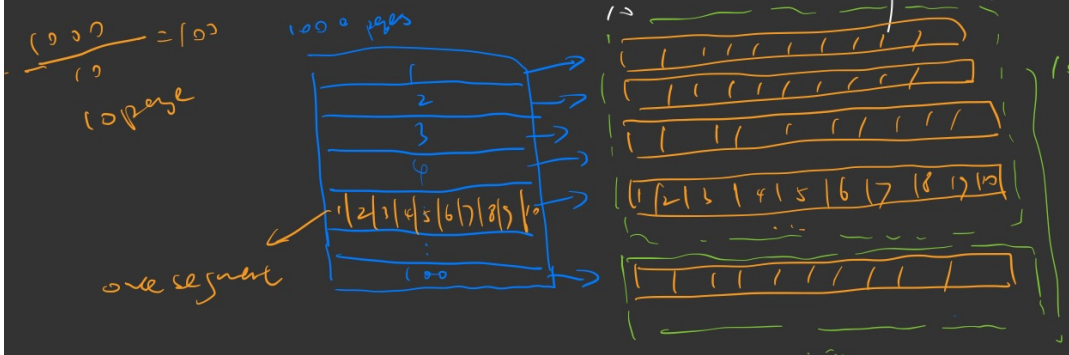
R 1000 pages



S 10,000 pages

consider sort R. ① divide page into segments of 10 pages.

$\frac{1000}{10} = 100$ segments (to make analysis easier)



② merge step: Assume merge 10 segment a time.

iterate 2 times

100 segments, 10 pages each

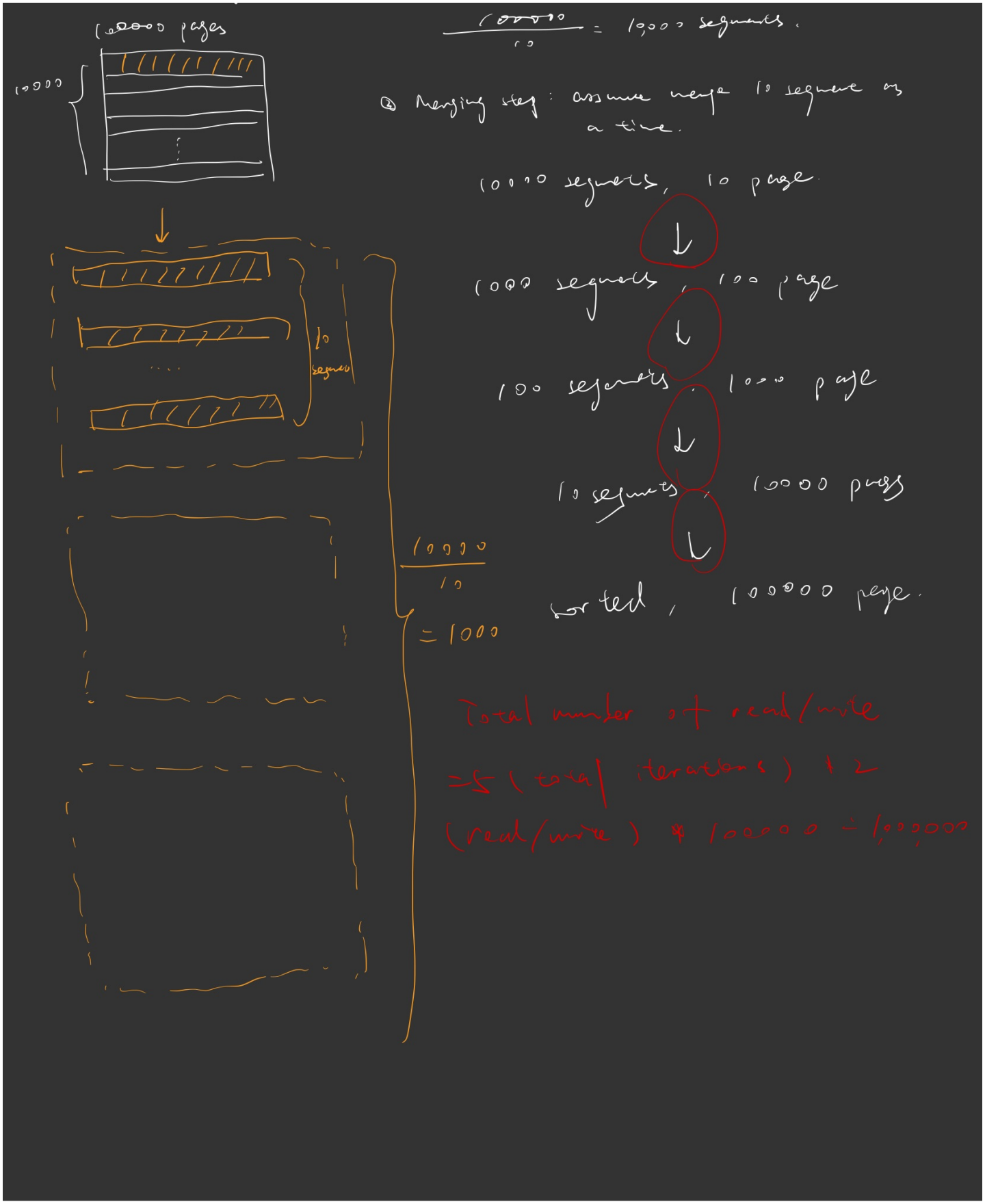
$\frac{100}{10} = 10$, 100 pages each

sorted, 1000 pages

③ Total number of read/write: 3 (total iterations) * 2 (read/write) * 1000

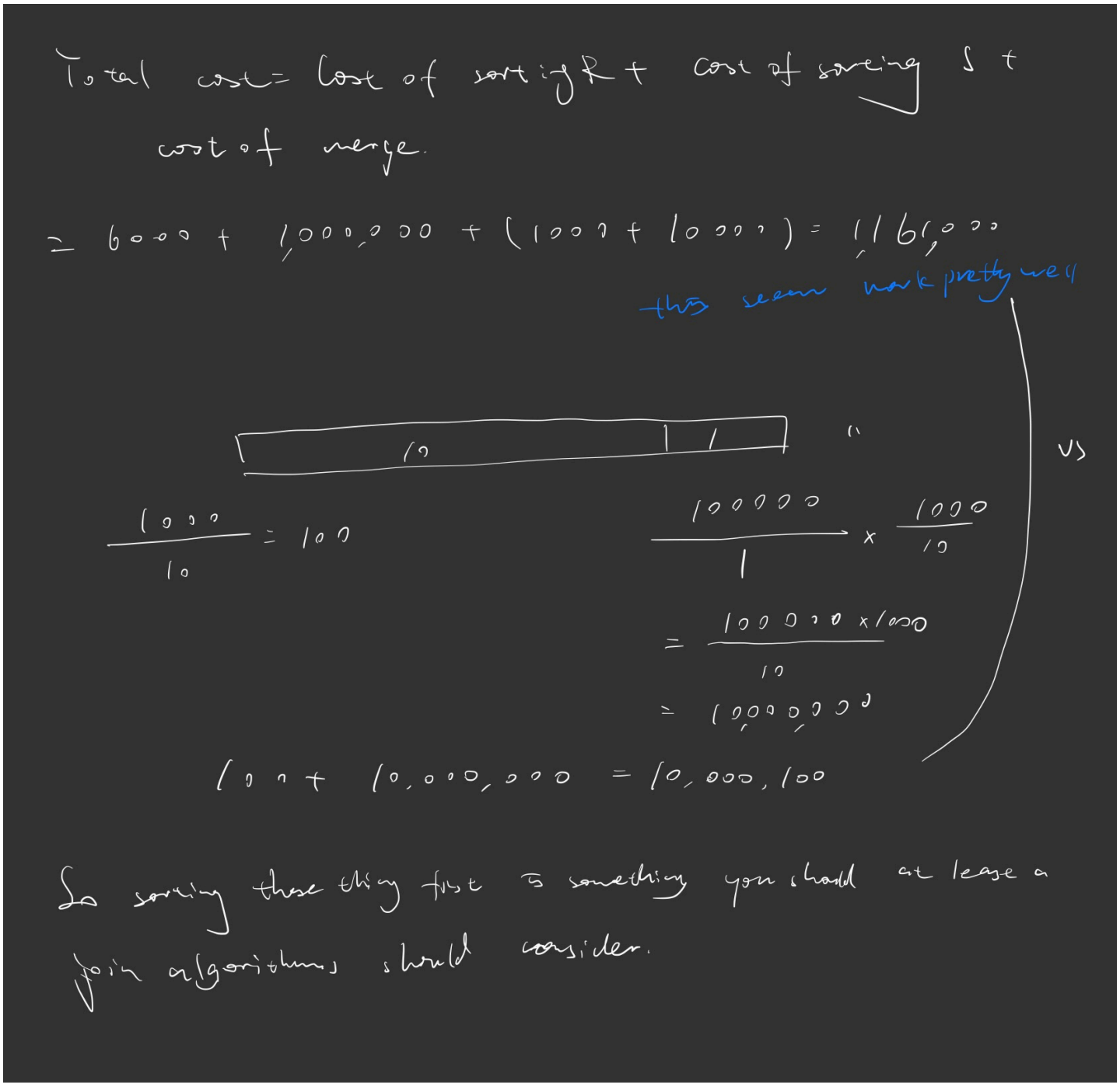
= 60000

Consider sorting S ① Divide page into segments of 10 pages



- Example: Consider joining to tables R (1000 pages), S (100000 pages)

- Assume we have 11 pages of buffers
- Total cost = Cost of sorting R + Cost of sorting S + cost of merge = 60000 + 1000000 + (1000 + 100000) = 1161000 pages



> What is the weakness of a solid state drive? About update. And you have to write you have to do quite a lot of update temporary tables. Databases have to balance between the two things. Are you willing to do a

lot of updates to kind of shorten the lifespan of a solid state drive? Maybe in ten years when the solid state drive technology, a much better may be worth it.

Now a good exercise. We have 101 pages of buffer where there is still that much work. Something you might want to think about just to the back of the envelope calculation and ignore the seek time and see if this is working.

so the question is, what if I have a non clustering index now?

```
select *
from A, B
where A.a, B.b
```

Let's say we have a non-cluster index on B. I will still have to go to table to retrieve all the other attributes. And because even though is sort of in the index is not sorted in the table. So that means the ordering can be quite arbitrary there. But sort-merge is definitely an option that a database system should think about. Sorting is certainly not cheap, even though it may be cheaper than doing so. So there is a alternative has shown.

15.5.5 Joins – Hash joins

- $R \bowtie_{R.a=S.b} S$
- Build a hash table on the file for attribute a and b, using the same hash function for both tables
- Now only tuples in the same buckets in the corresponding table can join together

How can I use hash table to help me to do a join?

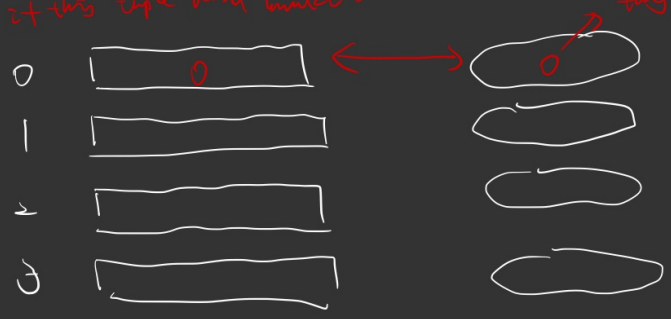
There's nothing stopping you to use the same hash function on both attributes, but one thing that's $logn$ is.

h

I'll first hash table R. R.a

the attribute a.

Let's assume we break into 4 buckets.



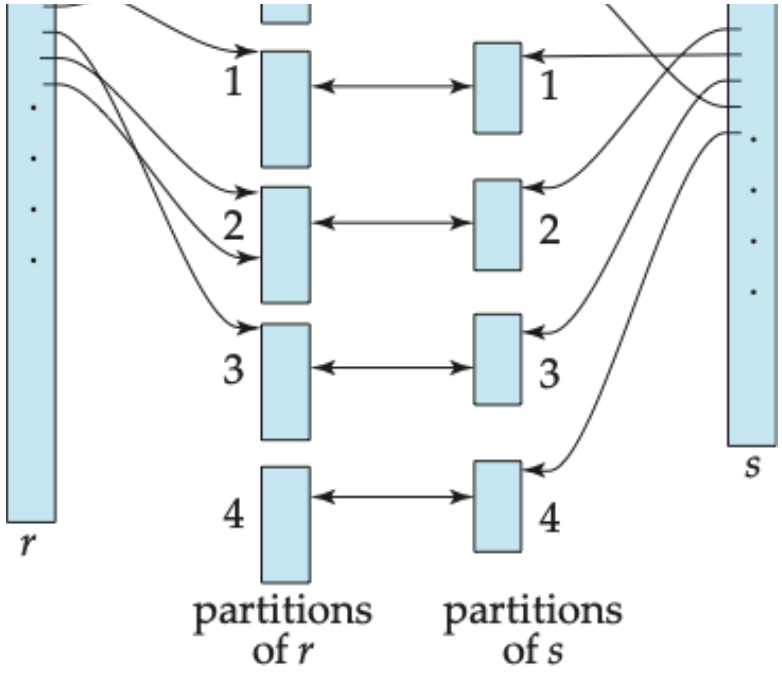
I'm practically breaking table R and table S into 4 parts each.

We only need to join the same buckets.

If they have the same value, they are going to have the same hash value.

But does it mean that every tuples here will match every tuple here?

A: Not quite. Because you maybe it only a hash function. They may be different value that can be hash to the same bucket.



- Now each pair of partition can be join by using any algorithm
  - If **one** of the partition is small enough to fit in the buffers, do a nested loop

    > one of the segments is small enough, then I can stop the recursion. I don't need both of them to be small enough.

  - Otherwise, one can recursively apply hash join – using a different hash function for each recursive call, until one of the partition is small enough

    > So you do need to have quite a few hash function in your hand, and you also want to try to make sure those hash functions are independent of one another. That can be tricky. I'll give you that. This is a bit tricky to estimate.

    > Can you think of advantages and disadvantages of each method? Now assume that we have enough hash function available.
    >
    > What do you think of a big deal of one not two?

(R 1000 S 10000)

(R 1000 S 1000000000)

Obviously the second one has to cost more time than the first one.

But the number of iteration is the same. Because you only need the smaller table to fit in memory. Then you can stop the recursion. It can save less how large the table. Type linear time.

Let's think about sort-merge. You have to sort both table.

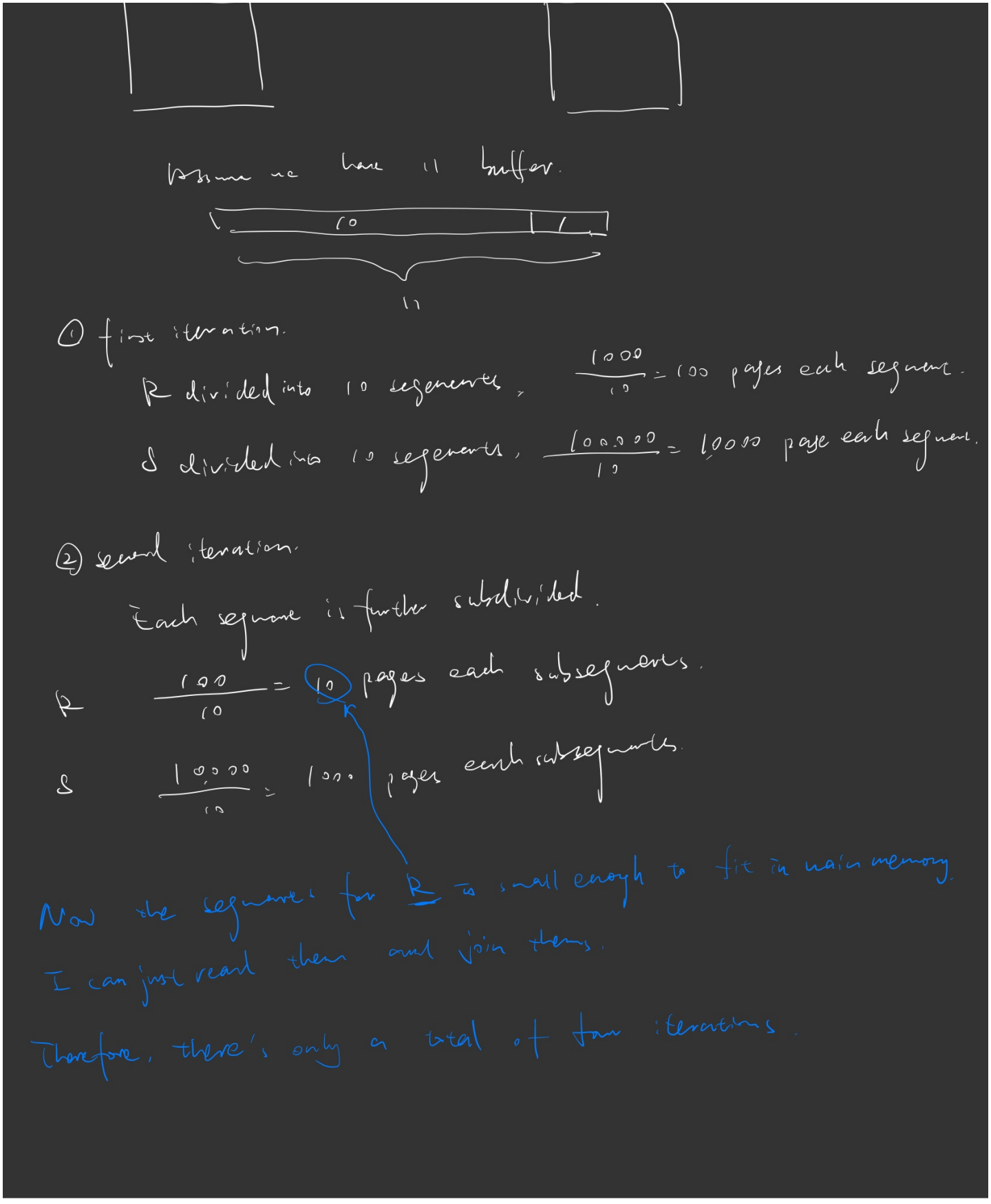
So if the two tables are drastically different in size, then Hash Join is quite attractive. at least compared with sort merge.

You can control the hash function, but you can't control the data distribution.

- Cost

- Harder to estimate
- Depends a lot on data distribution (as a skewed distribution will lead some segments very long)
- However, potential advantages
  - As long as one table's tuple behave nicely with the hash table, it should be fine
  - Even if distributions are skewed for both tables, it is possible that they complement each other (i.e. for a hash value, one table may have a lot of tuples, but the other may have little)

- Example: Consider joining to tables R (1000 pages), S (100000 pages)
  - Assume we have 11 pages of buffers
  - Assume all hash function evenly distribute the tuples for both tables
  - First iteration:
    - R and S divided into 10 segement (100 page for R, 10000 page for S)
  - Second iteration
    - Each segment is further subdivided (10 page for R, 1000 page for S)
  - Now the segments for R is small enough to fit in main memory
    - Nested loop for each pair
- Running time = 4 * (100000 + 1000) + 100000 + 1000 =505000
- Better than sort-merge



Ioins – hash-ioin vs. sort-merge

- Comparing sort-merge and hash-join
  - Hash-join is better than sort-merge when there is a large different between number of pages between the tables
    - The number of iteration is dominated by the larger table in sort-merge (need to completely sort both tables)
    - But is dominated by the smaller table in hash-join (need to recursive call until ONE of the segment is small enough)
  - Sort-merge has more predictable performance
    - Hash join's performance depend on how the hash function performs

      > sorting algorithm is not affected by data distribution

  - Sort-merge has the output sorted by the join attribute
    - May be important (see later)

      > This is better than hash join

> ```
> select *
> from A,B
> where A.a = B.b
> order by A.a
> ```
>
> Now, if you do sort merge, this is done for free.
>
> Hash doesn't guarantee order.

> ```
> select *
> from A, B, C
> where A.a = B.b
>       AND B.b = C.c
> ```
>
> What if I have this query?
>
> `A.a = B.b`: if I have resort for this join, then the next step when I do sort merge, I only need to sort one table. Now the cost of sort merge for the second join become much smaller. And this you have the factory. So that's why even though for individual join hash join are usually take before merge. Most database system still allow merge to be an option. Because by nature order by a they also very common in terms of query.

15.6.2 Projection

- Seems straightforward – just picking the corresponding values out of the tuple
- But how about SELECT DISTINCT?
  - Need to remove duplicates
- Much trickier than you think
  - Data too large to fit in main memory...

```
select A.a
from A
```

Just read the table. Just out of a. But remember, sql do not remove the duplicate before. Relation argument do, sql doesn't.

```
select DISTINCT A.a
from
```

What if user required to be distinct? Now it is trickier than you think because your table is too large. If the table cannot be the main memory, how do you know when I read that to pull the value has been returned or not? But what if your table is too large that you can't begin to make memory? And how do you remember the value that you have been returned? Things is trickier than you do anything.

In SQL projection is easy because in sql does not require you to remove duplicate unless the query explicitly say so.

```
select age
from student
```

There's no requirement for removing duplicate, you just read the tuple get the value and that's it. If there are 500 students of age 19, then you output 500.

Table Car

Car	Color
Car1	red

Car2	Blue
Car3	Red
Car4	Blue

```
select color
from Car
```

If the table has 200,000 tuples, I don't care where all the cars are where you can get 200,000 tuples. That's the price you're getting by doing that quickly. So they are situation which I really only need those color and I really want to remove duplicates. So sql do provide you with this operation. So now you qurery 200,000 tuples and the only two color red and blue return two tuples. But as I say so if there's no distinct cost, then projection is nothing more than just every tuple. Pick the attribute you need and output it. That's nothing to worry about.

However, if I have this thing, things become tricky. Let's read the first tuples, remember that there's a red. Let's read the second tuple, let me check, has been here before?

```
select DISTINCT student.ssn
from Advisor
```

Obviously it is meaningless. The SSN has not been appear. But how do you know that the SSN has not appeared yet? You have to store all the SSN that has appeared. But if the table is so large, you cannot even store all the SSN and you finally in main memories. What happened? See the problem here. But all do is hey, let me check whether that value you have been has appeared before. But what if there are so many variables because the table is so huge? Let's say that you cannot even store all the things you have seen in main memory. Remember, you can only compare things when they are in main memory. If you don't have space in main memory then what? You have to kick something out.

So we can acutally sort the tuples on the attribute to be return. Why does that help? Because the duplicate always be next to one another. So all I need to do is to keep track of the last well being output. Then next tuples is that duplicate. It must appear right after. Then you don't need extra space to store it.

- Use external sort
    - At each step, eliminate duplicates for a segment before writing on the disk
- Hashing can be used instead

> But you have to do some hash Joinish. Once again, you break the table down into smaller tables, into smaller segements. Each segements might contain values of the same hash. You may have to rehash it again, again and again until you until the table is smaller. So you can do either one of those.

15.6.5 Group by + Aggregation

- **Aggregation** can be implemented in a manner similar to duplicate elimination.
  - **Sorting** or **hashing** can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - Optimization: **partial aggregation**
    - combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
      - When combining partial aggregate for count, add up the partial aggregates
    - For avg, keep sum and count, and divide sum by count at the end

```
select dept, avg(gpa)
from student
Group by dept
```

In one sense, this looks like a projection with duplicates. You have to group things with the same value together. That step can be done by sorting or hashing. But interesting, if you have typically, because once you form group. Now remember you sql syntax. Once you use `group by` from group really what can you return? You obviously can return the attribute that useful grouping. Other than that you really can only return aggregates.

```
select dept, SSN
from student
Group by dept
```

For example, this is not allowed, you cannot do that in sql. Once you group the tuple thing together in one group you cannot output individual tuples value. That means once you've done the group by. These are all aggregate values. Now, depending on what your systems support. All systems support the simple ones, like `sum`, `avg`, `max`, `mean`. Some system will allow you to do more fancier thing like standard deviation, square whatever. Remember, all we need to return are group values.

So let's say we do sorting. What we can do is that while we are creating segments that are sorted, we already we call the aggregate words.

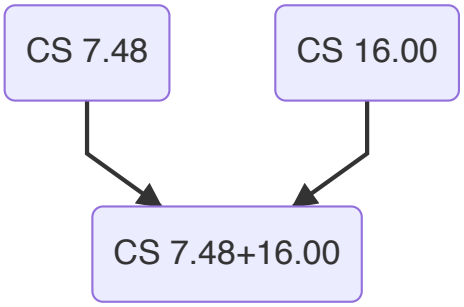
```
select dept, sum(gpa)
from student
group by dept
```

```
group by dept
```

Let's make it easier, that's a sum I want this sum up gpa. Not very meaningful, but I'm using if for illustration. So let's say I do some kind of sorting. And remember, if the table is too large, what kind of sort can I do? -> merge sort. So what should I do in the beginning. Let's split into small segements, and then we sort each segment. Notice that once if I saw ech segment do I really need to keep individual tuples once I create a segment?

Because firstly, once again, an individual tuple value is not to be output. So keeping SSN and all these staff quite meaningless.

Number 2, if I want to calculate the sum, let's say in this segement we have four guys from the cs department and the sum of their gpa is 7.48



But once you've done the initial step, you can immediately collect the aggregate value. Now the good news is what the aggregate value tends to be smaller than tuples. So even though say this is one segment is one segment, if you combine the tuples and form the aggregate value that you need, you have probably too many of these things into one segment. And then you continue to do the emerging step.

Once again, if I have two groups, say the first group, the CS avg gpa is 7.5, the second CS avg gpa is 9.4. How do you calculate the overall average of all the avg? Can I do 7.5+9.4? ❌ . If A has 2 student, B has 4 student. Or A has 4 students, B has 3 students, the average will become quite different. So we need to keep both the sum and the cap in order to calculate the average. But hey, that's one more number, so it's not too bad.

median, unfortunately, is something you cannot get away with it. You really have to do a merge sort on the whole table. Get all the tuples in order before you copy the median. you should also keep the count to make it easier. In the final step, we know how many tuples do I need to count before we get the median? But median is something that you cannot get away and many databases dump to provide you to provide

a user to hey, get me the median of each group. So this doesn't always work. And whether it works depend on what is the aggregate function you use and depend on the aggregate function we use. We may need to store more or fewer information. For example, if all gets cut, then you really just need to store the

> cut. If I want to aggregate every then I need to keep each other car in the sun if I want to do standard deviation. Then you need to keep sum square of sum.

> And to make thing more interesting for group by. That's a having costs. What is having means?
>
> Select which group do you select. Not which tuple. Tuple select is done in the well cost. And you select your tuple before you grouping. You do the well cost first, then you do the group.

`Having`:

https://blog.csdn.net/qq_37634156/article/details/120055284

> ```
> having count() > 12
> ```
>
> So that also can make things a bit dicier. For example, count greater than 12 clients.
>
> ```
> having count() < 12
> ```
>
> If you have aggregate conditions, which one do you think will run faster?
>
> `<12` faster. Why?
>
> If you have count less than 12. Once your group has 12 elements, You can kick it out and never consider it again. If you have concrete and 12, you really have to wait till all the segments to be embedded in it before you know whether you have actually more than 12 elements. Because you may be so unlucky that all the tuples that satisfy the condition is in the last segment that you constructed. You have to wait to the bitter end.
>
> If you cannot less than 12, you less than 12 tuples, once you got a 12 tuples, you can stop considering. So all the things build into mixing thing tricker that you need to learn how to do it. Query optimize it to figure out how to do it.

Union/Intersect/Except (set operations)

> We use less a little bit often in sql. Doesn't mean that we are not using them. Except it's certainly something that people used before. Because if you don't use except you will have to use quite complicated nested query, which you probably don't want.

- **Set operations** ($\cup$, $\cap$, $-$): can either use variant of sort-merge, or variant of hash-join.
- E.g. Set operations using hashing:
  - 1 Partition both relations using the same hash function
  - 2 Process each partition i as follows.
    - 1 Using a different hashing function, build an in-memory hash index on r_i.

- 2 Process s_i as follows
  - $r \cup s$:
    - 1 Add tuples in s_i to the hash index if they are not already in it.
    - 2 At end of s_i add the tuples in the hash index to the result.

```
select name
from student
where age >= 19
INTERSECT
select name
from student
where dept = "MATH"
```

https://www.1keydata.com/cn/sql/sql-intersect.php

What is this query supposed to return?

All math department students

The name of all students from Math department at least 19 years old.

Is query correct? Does this return to tuples that you want? The answer is not quite. Why not?

Is name the key of students ? No.

Let us assume we don't care about Duplicate. You can have two student of the same name such that one of them is 19 years old. But Philosophy Department and the other guy is the math department, both 18 years old. Will that query return that student?

David Lee 18 years old / math department

David Lee 28 years old / philosophy deaprtment

The first query will return David Lee, and the second query will also return David Lee.

Remember, all I'm getting is name. I don't output any other attribute. So there's a David Lee in here.

```
David Lee
David Lee
```

All I do is intercept. So these two are the same. So sql actually return. But there's no David Lee who is actually in math department. So be careful where you use set operations. Be very careful. I'm not saying that you shouldn't use it.
Things like that are not exist. But you have to be careful about these potential problems.

Let's assume we actually have a query. Everything is correct. Now how to execute.

So what is intersection means where you have to be incoming in both sets. So how do we execute? So we have one sql query, but we have the other sql you a query.

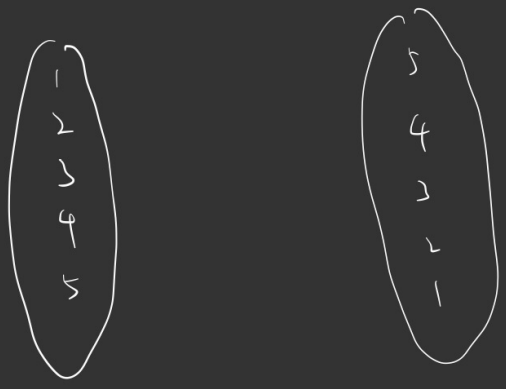
A easy way to think about intercept to essentially a joy such that every attribute would have to match. So you can do any join algorthms with the condition with that every attribute.

Hashing probably is a good choice here because you are looking at so many distinct value in your hash function really read that everything to the same bucket is either your lucky day or you really screwed up picking your hash function. So hashing is probably a good way to do this.

Union, because all everything is except that. And remember, you are by default. Do not remove the bucket. So actually there's no story to tell. Just concatenate everything.

Except: It is trick. It's not quite directly as a join. Just because two two poles are not equal. You don't want to return to both of them immediately. Why?

The second table return tuples 5, 4, 3, 2, 1



If I want to do EXCEPT. What tuples should I return?

Nothing.

But if I just do a join (and I are different. They are not the same. So should I return the one now? No. I have to look at all possibilities. So there is really relatively few cases of optimization because for each tuple I can not add content. I have looked at the whole set of tuples.

The best you do is sorted table so that once strings are not equal Then you know the one. So sorting is still a possibility, have you really do need a different program. You can not just rely on a join Operation. I can not intersection. Because intersection just a join.

EXCEPT is not a Join, because for each tuples, I have to make sure none of the other tuples match. If tuple A doesn't match with the first tuple that means nothing because it may match with the 17 tuples. It doesn't mean I should return. I have to wait to examine all possibility before I can be sure that I can return

> this to one. So that's why. That's why I say for intersection. There's really nothing too much I can do. Sorting is your best bet.

- Set operations using hashing:
  - 1 as before partition r and s
  - 2 as before, process each partition i as follows
    - 1 build a hash index on r_i
    - 2 Process s_i as follows
      - $r \cap s$

        1 output tuples in s_i to the result if they are already there in the hash index
      - $r - s$

        1 for each tuple in s_i if it is there in the hash index, delete it from the index.

        2 At end of s_i add remaining tuples in the hash index to the result.

        Difference in except: since operation is NOT commutative, restriction on which table can be "outer" loop.

        > $r - s$ is different with $s - r$

> One of the tricky thing about sql and in fact it's actually true whenever you talk about a logic operation negative information you can do it usually take a lot more time. Because if I want to say someone exists, I just need to find out the first exist and done. So there's a lot of optimization can do.

Outer joins

- Challenge: The unmatched tuples still need to be returned (with NULL value padded)
- Sort-merge:
  - Straight-forward: output also unmatched tuples during merge
- Hash-join:
  - Once again, restriction on what can be the “outer” loop

> But now, outer joy is relatively popular, what does older join mean, let's say a able to join?
>
> You first $A \bowtie B$.
>
> It's the tuples that does not match. You still return it with the other field being set to not. So if there is no match, you still have to return it.

> And that's why it makes sorting a bit more attractive, because once again, once you sort, you can just do a linear scan and a tuple is not find. The linear scan will reveal without it needing to go to all the way to the back. I keep you give you a key insight on how do we do individual operations and database system. They are coming out with more and more operations. That's how many of you know that sql have standards.

Data warehouse

OLAP: on-line analytical processing.

> I really don't care what other individual operation. I want to see trends.I want to see aggregates. I want to see groups. so I want to have additional operation in school that I can do all these things more efficiently. So sql is constantly adding operations. But the good news is that even though they are constantly adding operations, the overall framework of CRE optimization doesn't have to change. You just need to figure out how to implement each individual operation. But for example, many experts tell you what we call CUBE combat.
>
> And that is one reason why relational relational models stand the test of time.