SQL语句是声明式的，只说了它要什么样的答案，而没有说为了获得答案需要经历怎样的过程，这需要DBMS的优化器来决定用户的查询的计划怎样被具体地执行

通过DBMS的优化器自动地去对SQL语句进行查询优化，有如下两种方式：

Heuristics / Rules，启发式/基于规则

通过一些规则/变换的手段，优化掉用户的查询语句里面低效的部分

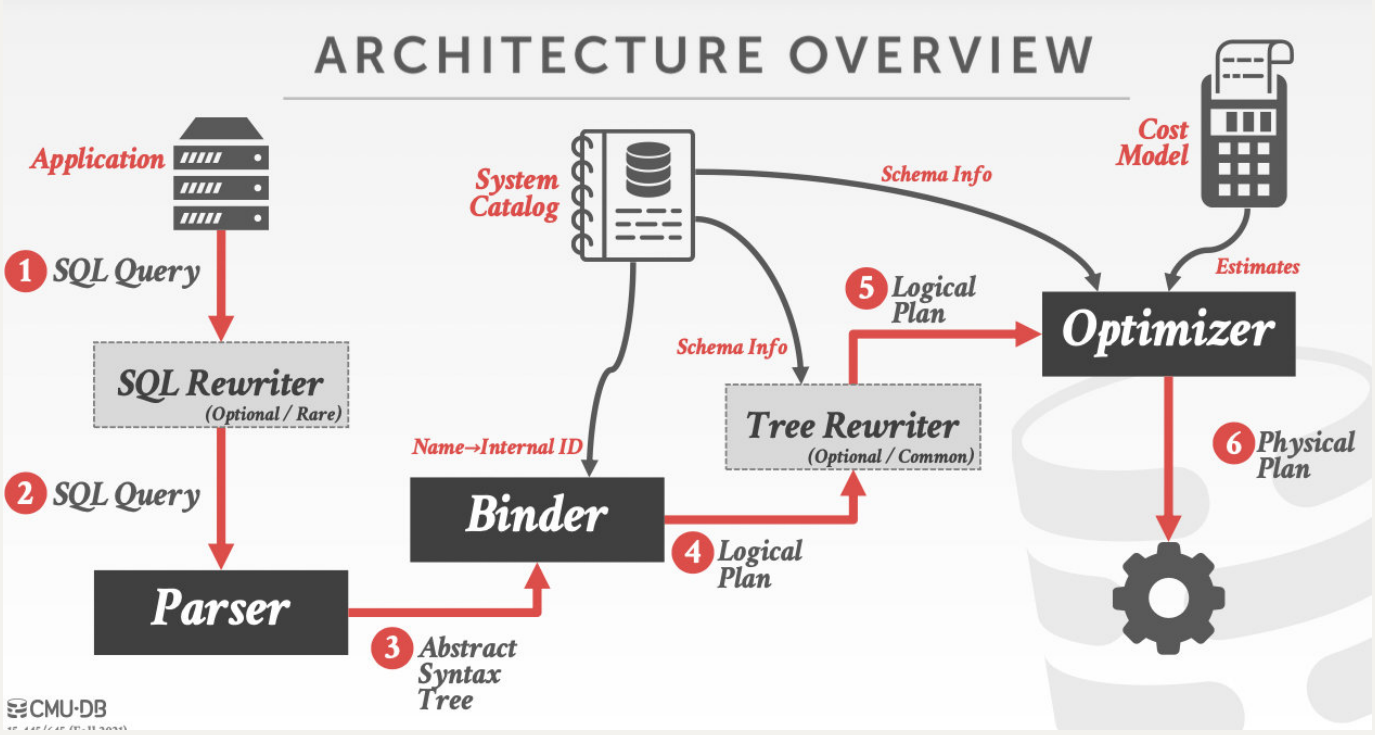
这个技术需要查询相关的元数据（catalog，比如说需要知道有没有给某个attribute构建索引），不需要检查数据本身（比如说获取数据的统计信息）

Cost-based Search，基于代价

需要构建一些代价模型，来计算查询如果按照某个计划去做的话相应的代价是多少，通过比较多个计划的代价，从中选出代价最小的计划去执行。而且为了计算出代价，往往需要知道数据库所存储的数据的一些统计信息（比如说数据的规模）

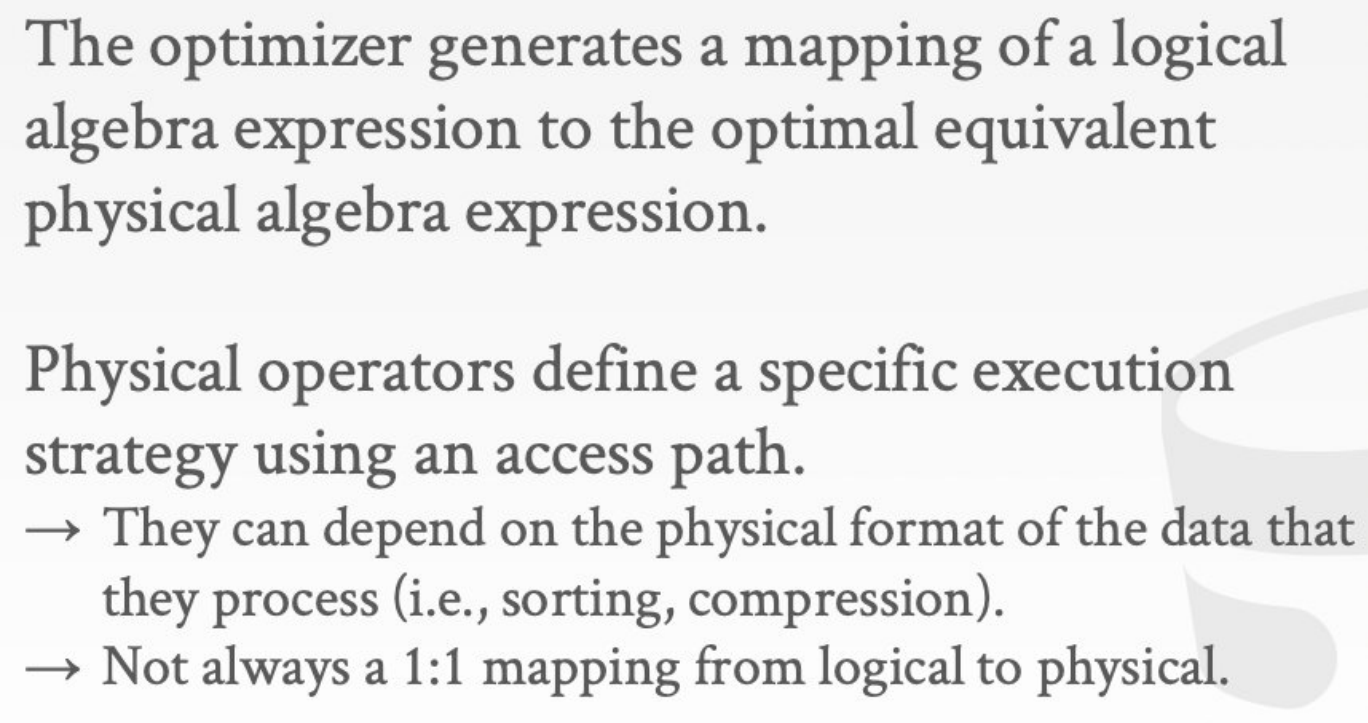
对于大部分DBMS来说，上述两类策略都会被用于查询优化，本**Lecture接下来会介绍上面的第一类-启发式的查询优化**

用户对DBMS的请求的执行流程如下



用户的业务会发出SQL查询语句，少部分DBMS会有SQL Rewriter这个组件，对字符串形式的SQL语句进行文本上的预处理（在字符串层面上做简单的优化），之后SQL查询语句进入Parser，Parser会把SQL语句变成抽象语法树，抽象语法树当中会涉及到库/表/列的名称，这些名称要和数据库系统元数据里面的库/表/列/索引的ID对应上，因此会有Binder（即连接器）把SQL抽象语法树中用户写的表名/列名/索引名转化成数据库内部使用的ID，并且这个步骤中会有检查：如果用户请求了一个不存在的表，那么就会直接报错。经Binder处理过之后的抽象语法树会被送入Tree Rewriter，这个组件大多数DBMS都有，它会输出一个标准的执行计划（比如说SQL语句里有一堆join操作，一开始的抽象语法树中的join的排布可能是乱的，Tree Rewriter会把所有的join排列成左深树，这个步骤也叫正则化），这个过程中也会查一些系统的元数据，Tree Rewriter输出的原始的逻辑计划是优化器进行优化的源头。之后基于规则的优化器（RBO， rule based optimizer）会查询一些系统的元数据来做优化，基于代价的优化器（CBO， cost based optimizer）不仅会查询元数据，还会查询相关的代价模型，根据代价模型去做优化，最后优化器会生成物理计划，被实际使用。

逻辑计划和物理计划的区别如下



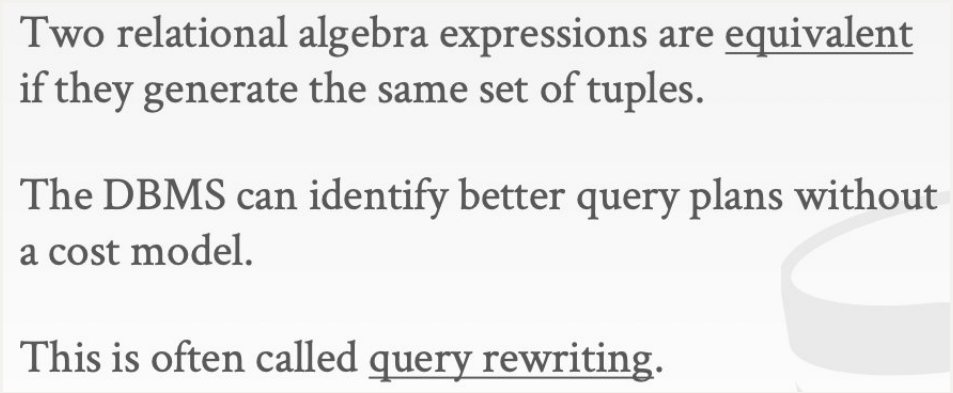
逻辑计划是关系代数级别的，物理计划包括了各个算子的具体执行方式，即物理算子（比如说join算子是用nested-loop join还是merge/hash join来完成）。并且逻辑表达式和物理算子未必是1对1的（比如说inner join可以对应nested-loop/hash/merge join这三个具体的物理算子）

查询优化属于NP-Hard问题（NP是非多项式的意思），甚至都不一定能得出最优解，是DBMS最难的部分。目前最新的研究中甚至引入了机器学习来辅助查询优化，但这样的人工智能模型有些时候是不可解释的，是一个黑盒

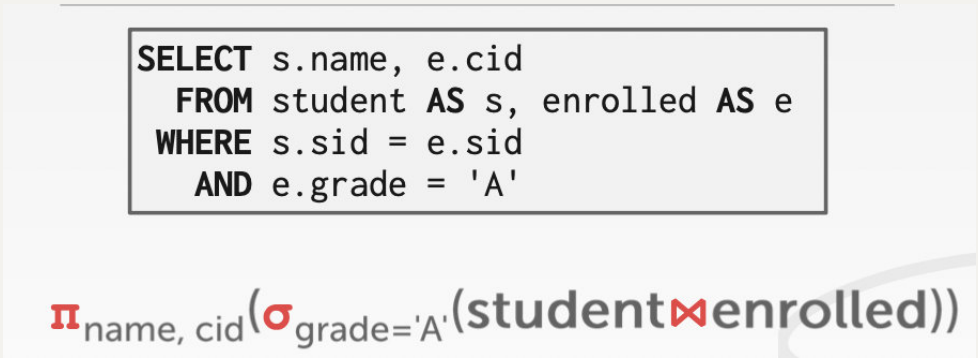
Relational Algebra Equivalences

关系代数表达式的等价

如果两个关系代数表达式所输出的结果集是一样的，那么它们等价

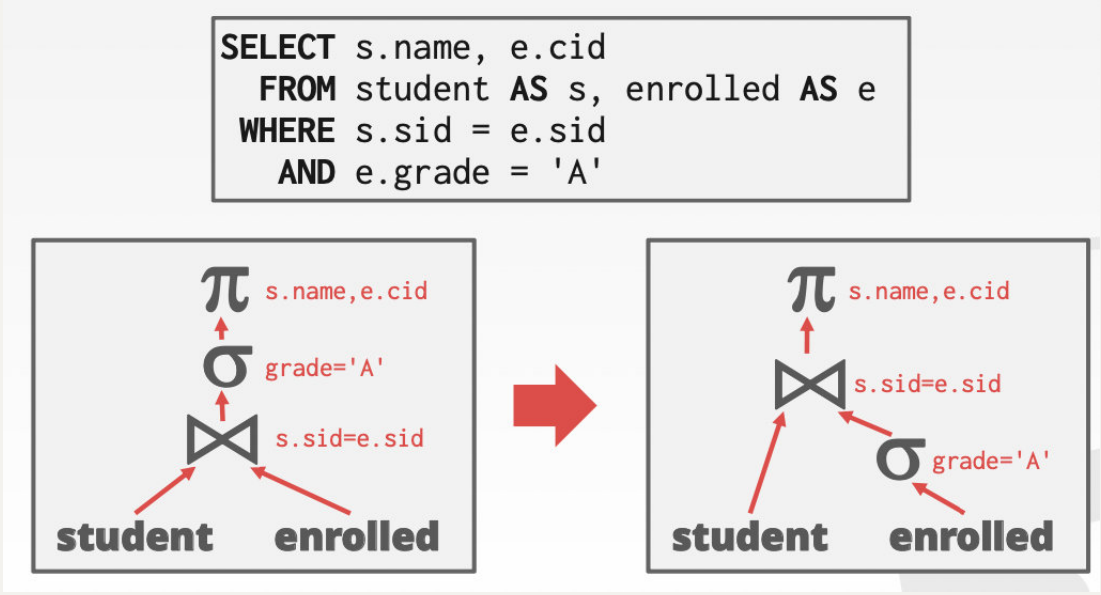


应用等价关系代数表达式进行查询优化的一个例子如下所示，



用户想对两个表连表之后做筛选，然后输出感兴趣的字段。上图中最下方是依据字面意思写出的关系代数表达式

我们可以把谓词下推，在join之前就进行筛选，这样的话就可以提升效率，并且输出的结果是一样的



也就是说，我们使用了下面的关系代数等式完成了查询优化

$$\pi_{name,\ cid}(\sigma_{grade='A'}(student \bowtie enrolled)) = \pi_{name,\ cid}(student \bowtie (\sigma_{grade='A'}(enrolled)))$$

还有一些其他常见的等价的关系代数表达式，如下所示：

Selections:

→ Perform filters as early as possible.

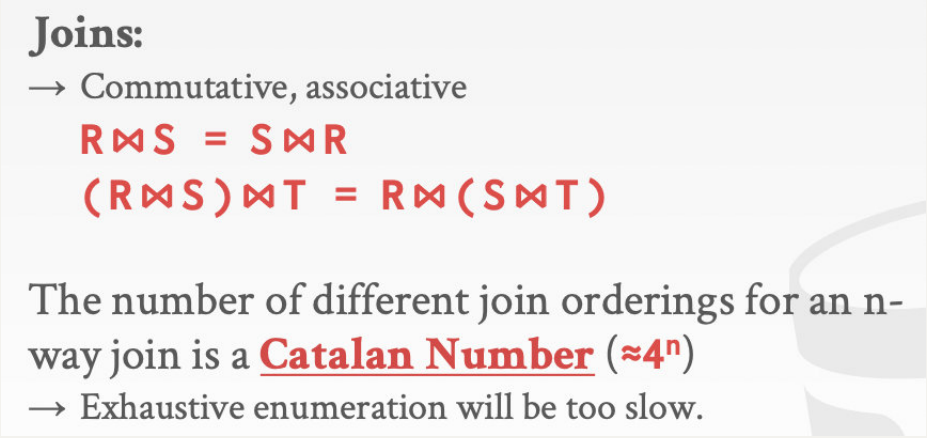
→ Break a complex predicate, and push down

$$\sigma_{p1 \wedge p2 \wedge \ldots pn}(R) = \sigma_{p1}(\sigma_{p2}(\ldots \sigma_{pn}(R)))$$

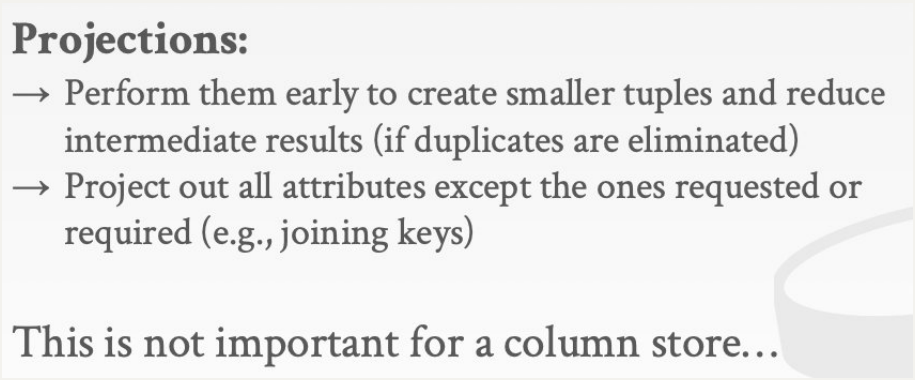
Simplify a complex predicate

→ (X=Y AND Y=3) → X=3 AND Y=3

join算子具有结合率、交换率，这会被用于查询优化



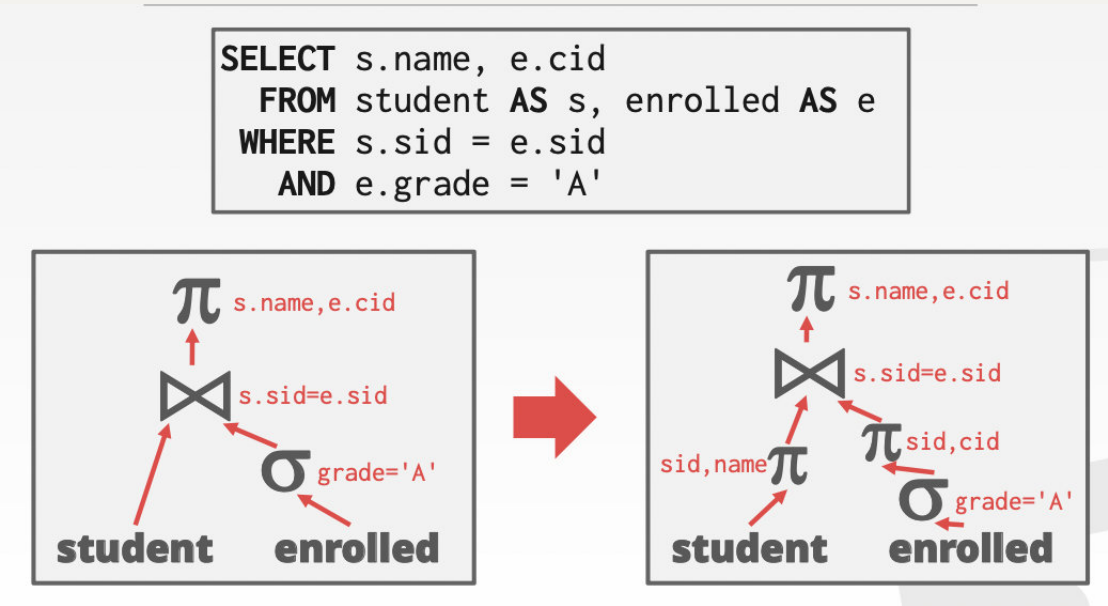
对于投影算子的实际执行，也有一些可以优化的点



- 类似于早物化，我们可以在执行过程中先执行投影算子，从而把无关的attribute对应的列删掉，让tuple和中间结果更小
- 类似于晚物化，可以先不管最后要输出哪几个attribute，只关注当前需要用的字段，等到最后投影的时候再回表现查，然后输出（假如投影之前要进行join，那么在join的时候只考虑join key，最后要输出的时候回表现查其他的attribute）

但这些查询优化策略对于列存储的数据库不重要，因为列存储的数据库永远都是最后进行对tuple的物化

结合实际场景分析，如下所示，我们可以把投影算子下推，实现早物化，只留下要投影的列和要join的列，这便可以提升上层的join算子的效率



Logical Query Optimization

逻辑计划的优化

> Transform a logical plan into an equivalent logical plan using pattern matching rules.
>
> The goal is to increase the likelihood of enumerating the optimal plan in the search.
>
> Cannot compare plans because there is no cost model but can "direct" a transformation to a preferred side.

我们要先设置一些优化规则，然后DBMS会对原有的逻辑计划进行模式上的匹配，如果原有的逻辑计划能够和优化规则匹配，那就将它变换成优化后的等价的逻辑计划

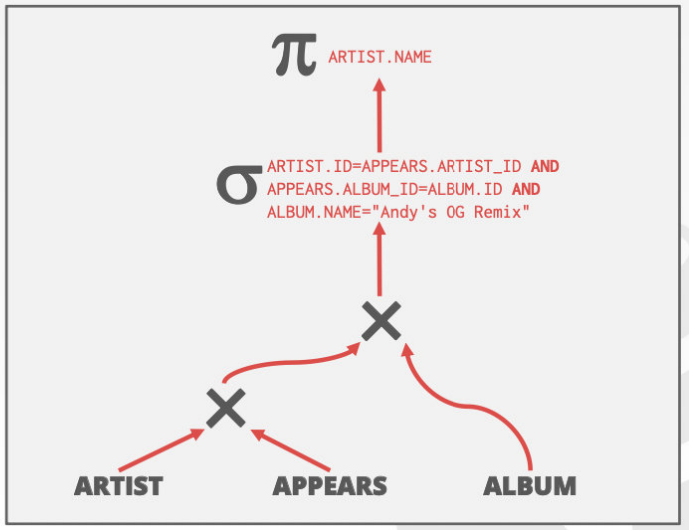
对逻辑计划的优化有如下的几个手段

- Split Conjunctive Predicates

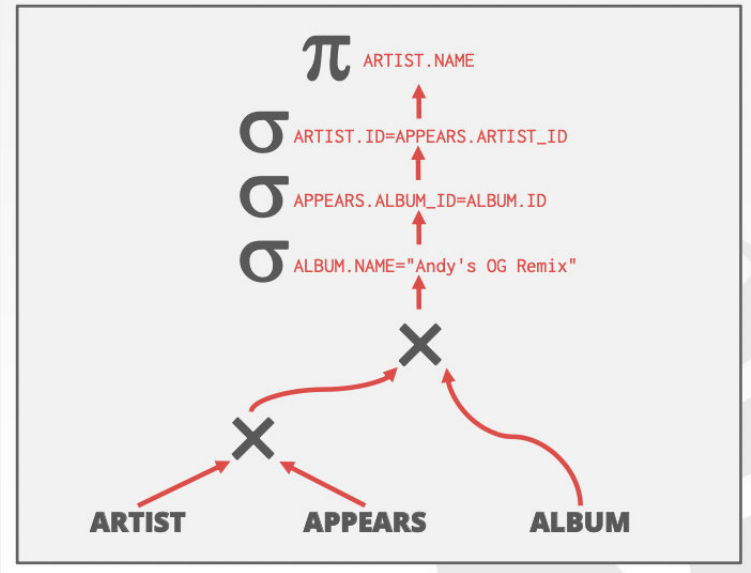
  把连接在一块的谓词分开，如下所示：

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

Decompose predicates into their simplest forms to make it easier for the optimizer to move them around.



->next->



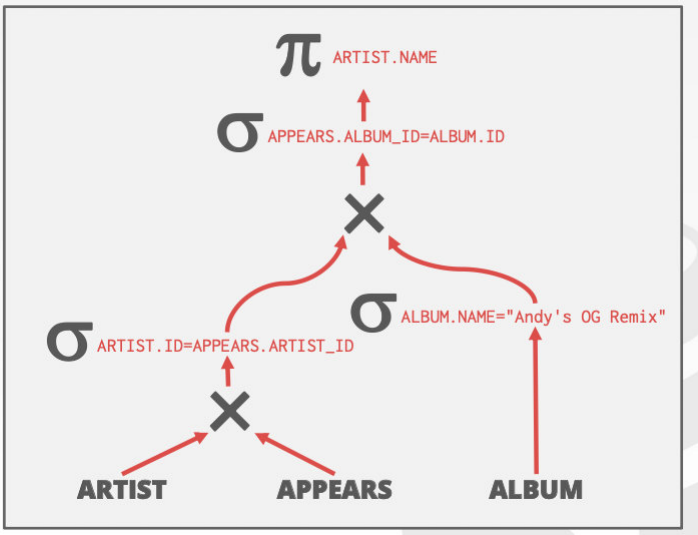
- Predicate Pushdown

  谓词下推，把谓词尽量往下推，推到越接近读表越好，这便可以提前筛掉一部分的数据，减少上层算子的负担

  实际操作如下所示（基于上面的图）

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

Move the predicate to the lowest applicable point in the plan.

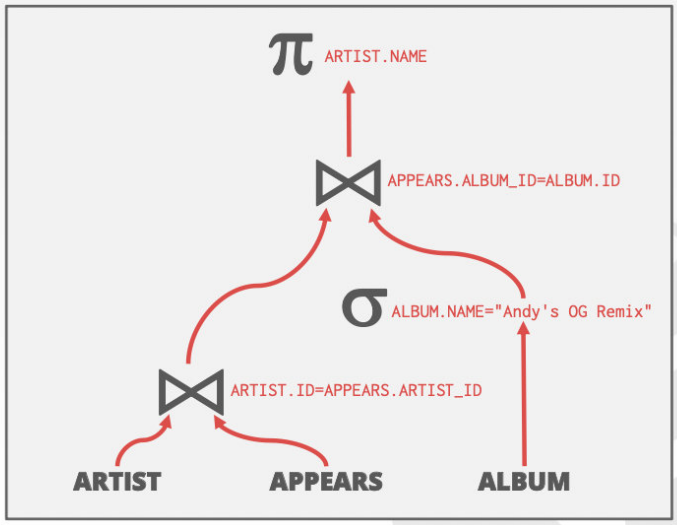


- Replace Cartesian Products with Joins

  根据相应的谓词，把笛卡尔积变成join，如下所示（基于上面的图）

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

Replace all Cartesian Products with inner joins using the join predicates.



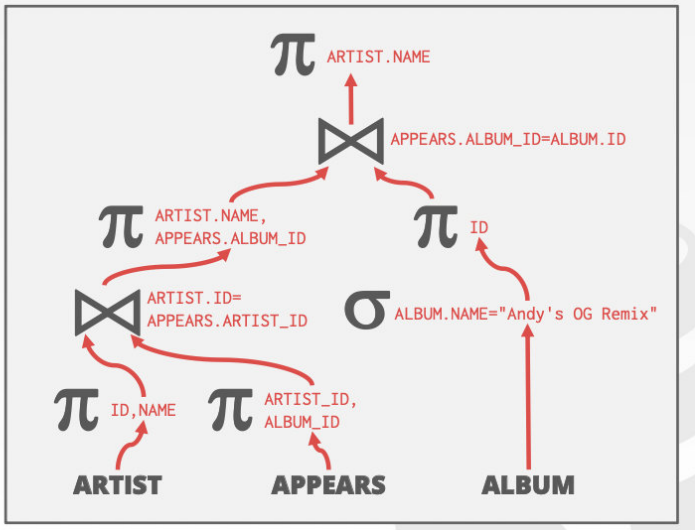
- Projection Pushdown

  投影下推

  还是继续分析上面的例子：我们需要的只是artist.name这个attribute，因此被传递的中间结果里必定有一些attribute是无用的，所以说可以提前投影，如下所示：

```
SELECT ARTIST.NAME
  FROM ARTIST, APPEARS, ALBUM
 WHERE ARTIST.ID=APPEARS.ARTIST_ID
   AND APPEARS.ALBUM_ID=ALBUM.ID
   AND ALBUM.NAME="Andy's OG Remix"
```

Eliminate redundant attributes before pipeline breakers to reduce materialization cost.



Nested Queries

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:

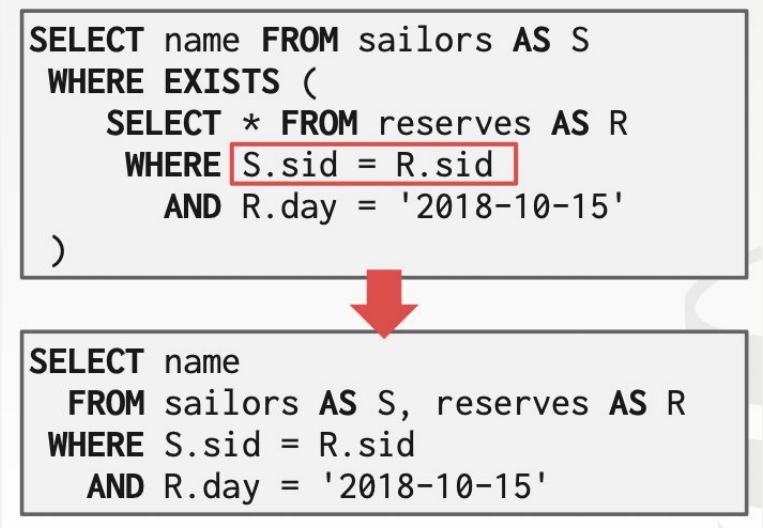
→ Rewrite to de-correlate and/or flatten them

→ Decompose nested query and store result to temporary table

SQL语句中经常会有一些嵌套的子查询，DBMS一般会用如下两大手段去优化它：

- 对子查询和外层的主查询进行rewrite，将它们写成一个查询

  结合如下场景分析：



原始的SQL语句里的谓词既涉及到了外层的主查询，也涉及到了内层的子查询，可以发现：实际上它想做的就是join，那不妨就直接进行两个表的join，写成上图下方的形式。这样的话接下来优化器只需优化这一个SQL语句，因此也更容易生成比较高效的物理执行计划

- 进行解耦：在一些复杂的查询当中，可能不太容易将内部的子查询和外部的主查询rewrite成一个查询，那么DBMS可以将子查询分离出来，先把子查询做完，把结果存放在一个临时的表里面，然后把这个临时的表带到主查询中。

  也就是说，这里的解耦是指：不让子查询嵌套在主查询里面，而是把它拿出来，提前执行它

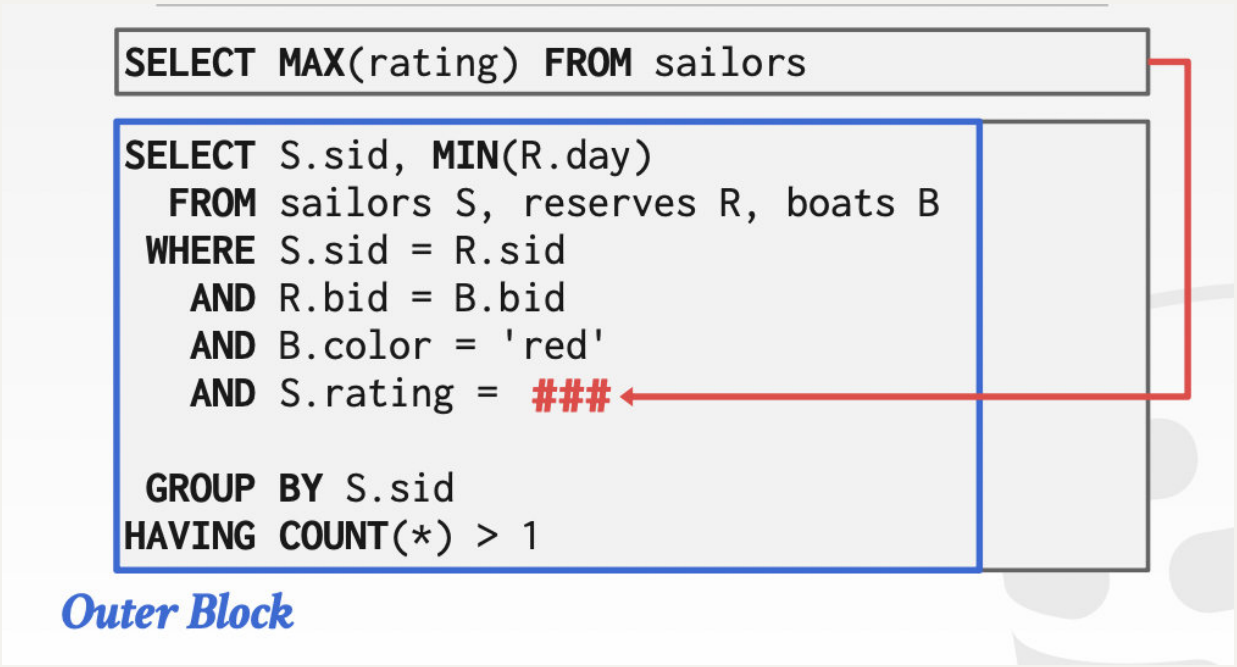
  > For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.
  >
  > Sub-queries are written to a temporary table that are discarded after the query finishes.

  结合如下场景分析，原始的SQL语句的谓词里面带有子查询，但这个子查询返回的值是一个常数，那不妨就进行解耦：先执行这个返回常数的子查询，然后把结果记录下来，否则每次使用谓词筛选时都会把这个子查询执行一遍，效率不高（这就类似于：与其多次进行重复的函数调用，不如把这个函数的返回值保存到某个变量里，然后使用这个变量）

```
SELECT S.sid, MIN(R.day)
  FROM sailors S, reserves R, boats B
 WHERE S.sid = R.sid
   AND R.bid = B.bid
   AND B.color = 'red'
   AND S.rating = (SELECT MAX(S2.rating)
                   FROM sailors S2)
 GROUP BY S.sid
HAVING COUNT(*) > 1
```

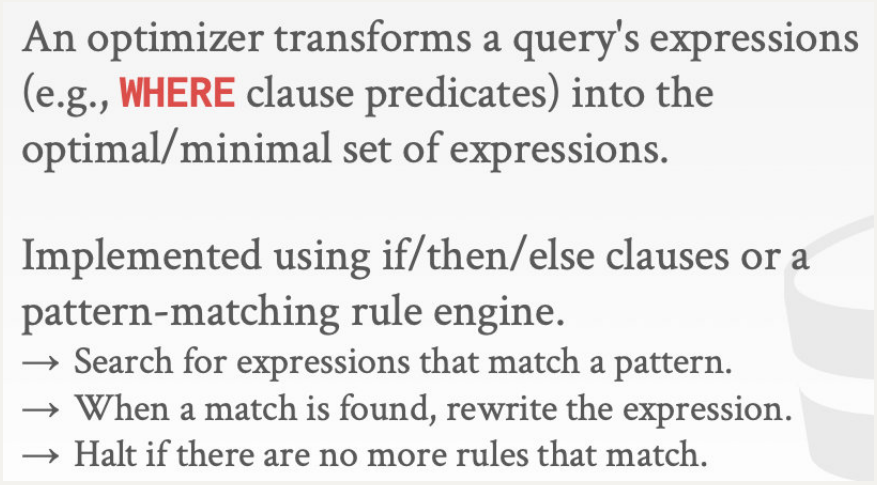
For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.

->next->



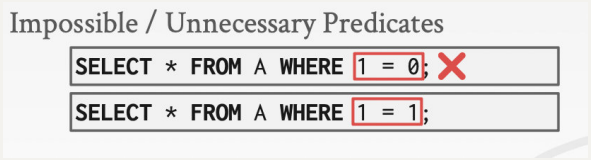
Expression Rewriting

表达式的重写

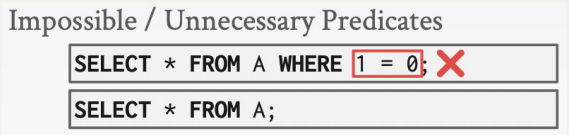


DBMS会通过人为设置的一些规则把查询语句的表达式（尤其是谓词表达式）重写，让它变得更加精简、高效

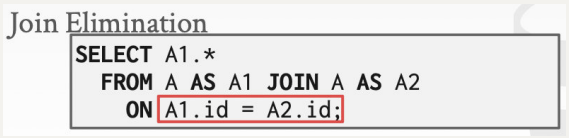
for example，如下所示，就像编译优化一样，DBMS的优化器可以把一些无效的谓词和join操作去掉



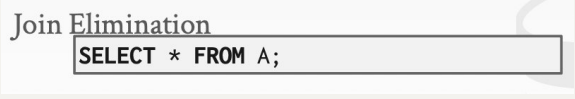
->next->



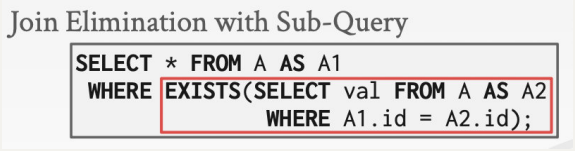
另一个例子：



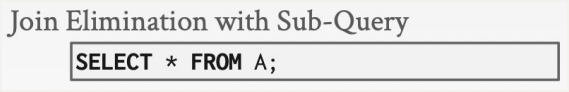
->next->



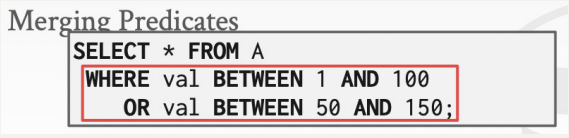
含有子查询的无效join也可以被消除：



->next->

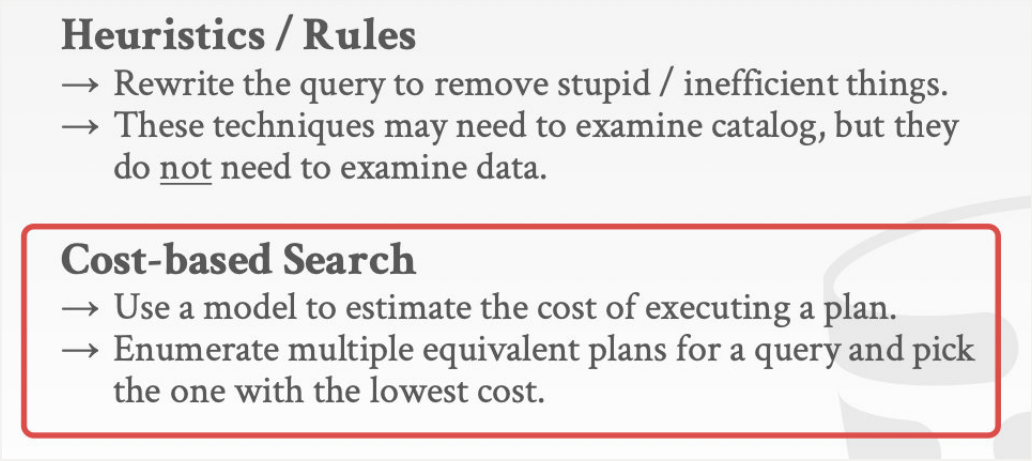


还有谓词之间的merge；



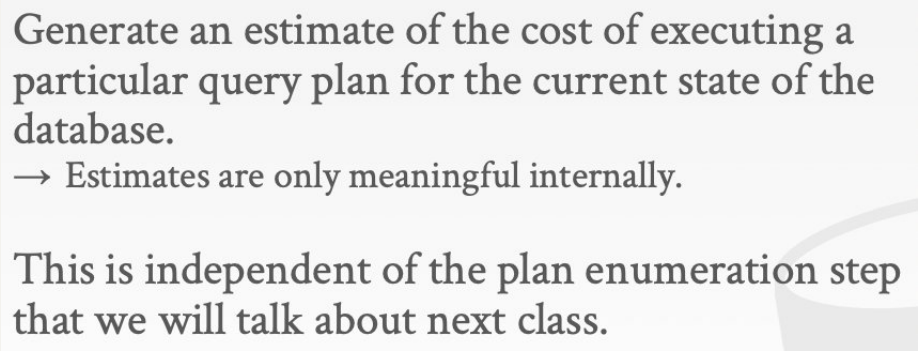
->next->

对于启发式的/基于规则的查询优化的介绍到此为止，下面将开始介绍基于代价的查询优化

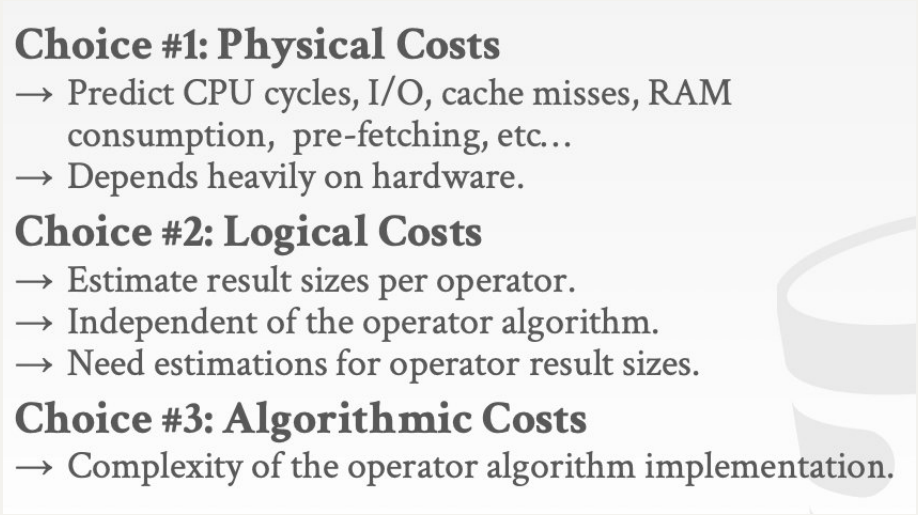


基于代价的优化器会根据当前数据库的状态估算出查询计划的代价，并且不同的DBMS的优化器计算出的代价值之间是不存在可比性的

并且估算代价这个步骤和计划列举这个步骤（plan enumeration，会在下一个Lecture介绍）之间是相互独立的

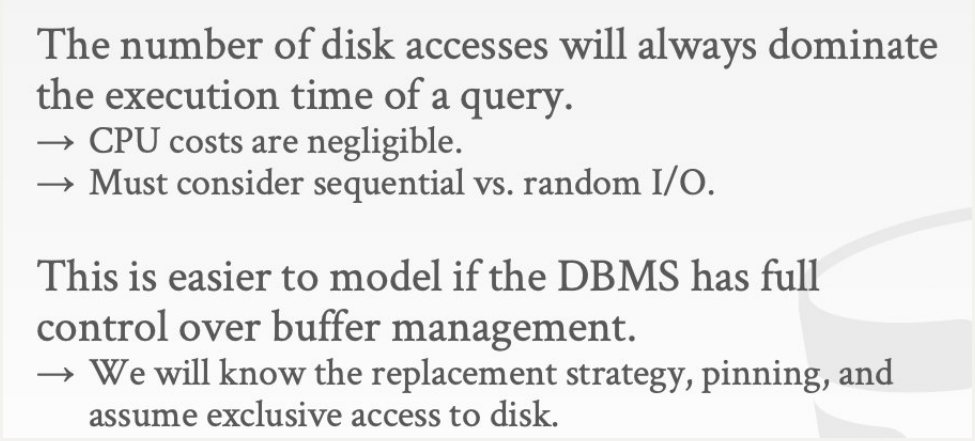


那么代价模型所估算出的代价来自于何处呢？包含如下三种：



- 具体的物理代价：上图所列举的那些，这和硬件的性能有很大关系（一般数据库一体机的优化器会更多地考虑这类代价，因为数据库一体机的DBMS了解硬件的全部具体信息）
- 逻辑上的开销：给每个算子大致估算它的开销（比如说hash join算子/table reader每处理100个tuple会有多大开销），而且由于是估算，所以这和算子内部实际采取的算法是无关的。并且优化器还需要估计每个算子所处理的数据的量，这便需要一些数据的统计信息
- 比较细粒度地估计算子的开销：分析每个算子的内部实现有几步，根据采用的算法的时间复杂度去详细地估计算子的开销

并且对于基于硬盘的DBMS来说，硬盘I/O的开销可能远大于其他类型的开销，所以要重点考虑，而且由于DBMS完全接管了缓存池，所以可以更细粒度地优化硬盘I/O开销



Ref/参考自 :

https://www.bilibili.com/video/BV1qR4y1W7v6/?spm_id_from=333.788