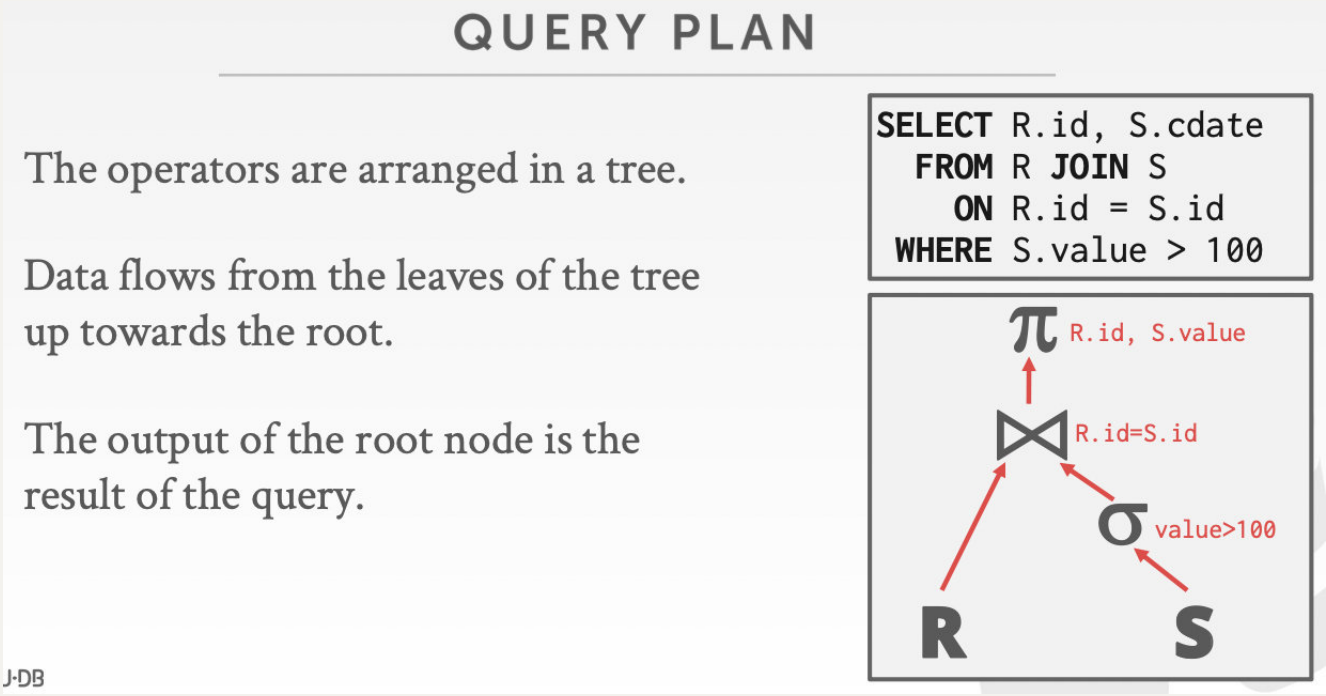
执行计划通过算子构成的树实现，数据从叶子节点往上经过不同的算子处理，最终从根节点输出

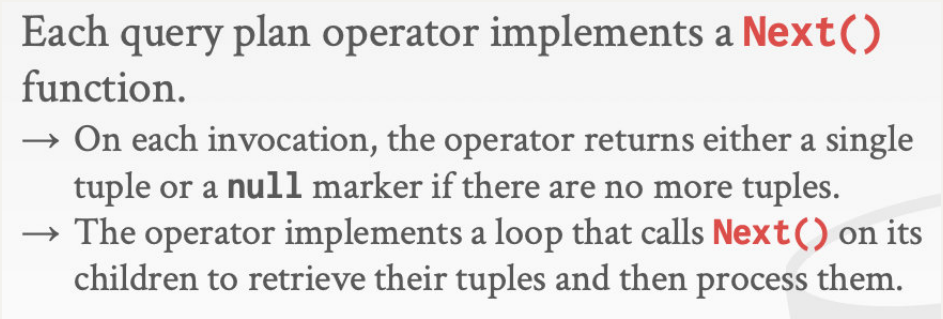


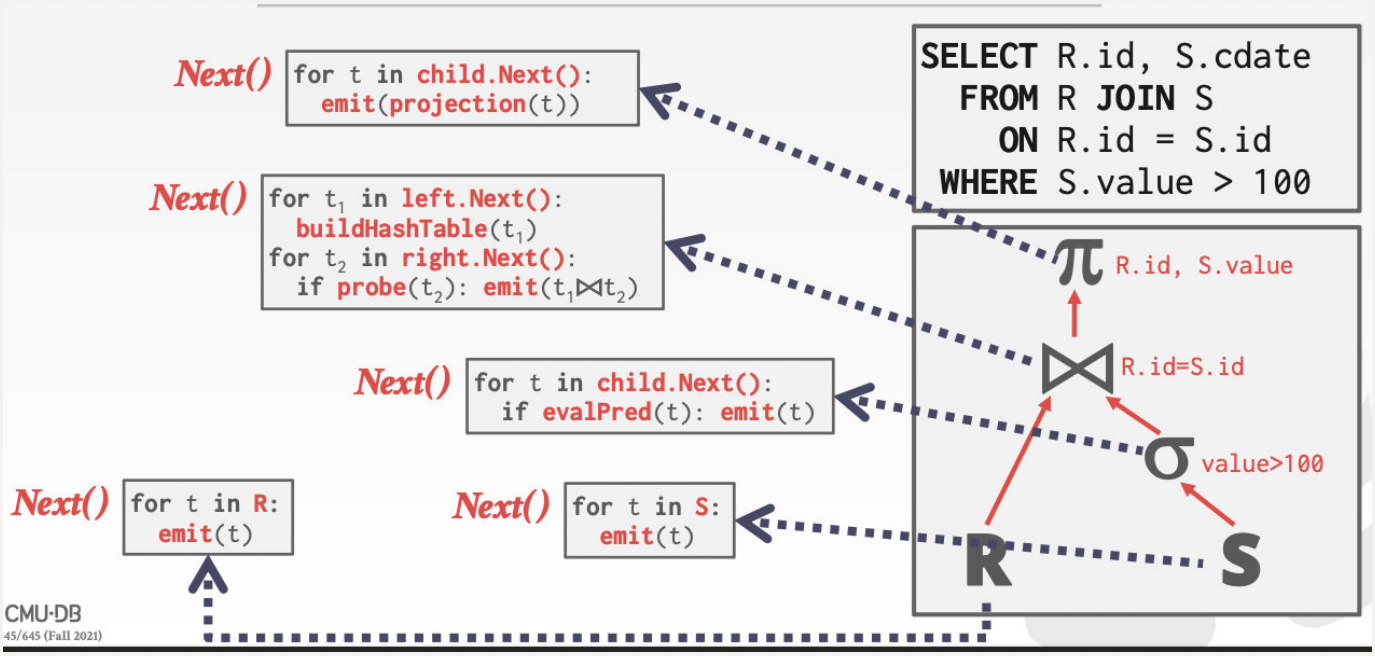
Processing Models

DBMS的查询语句处理模型定义了数据库系统将如何执行一个查询计划，根据不同的工作负载（如OLAP/OLTP），它在设计上有一些权衡，有如下三种常见的模型：

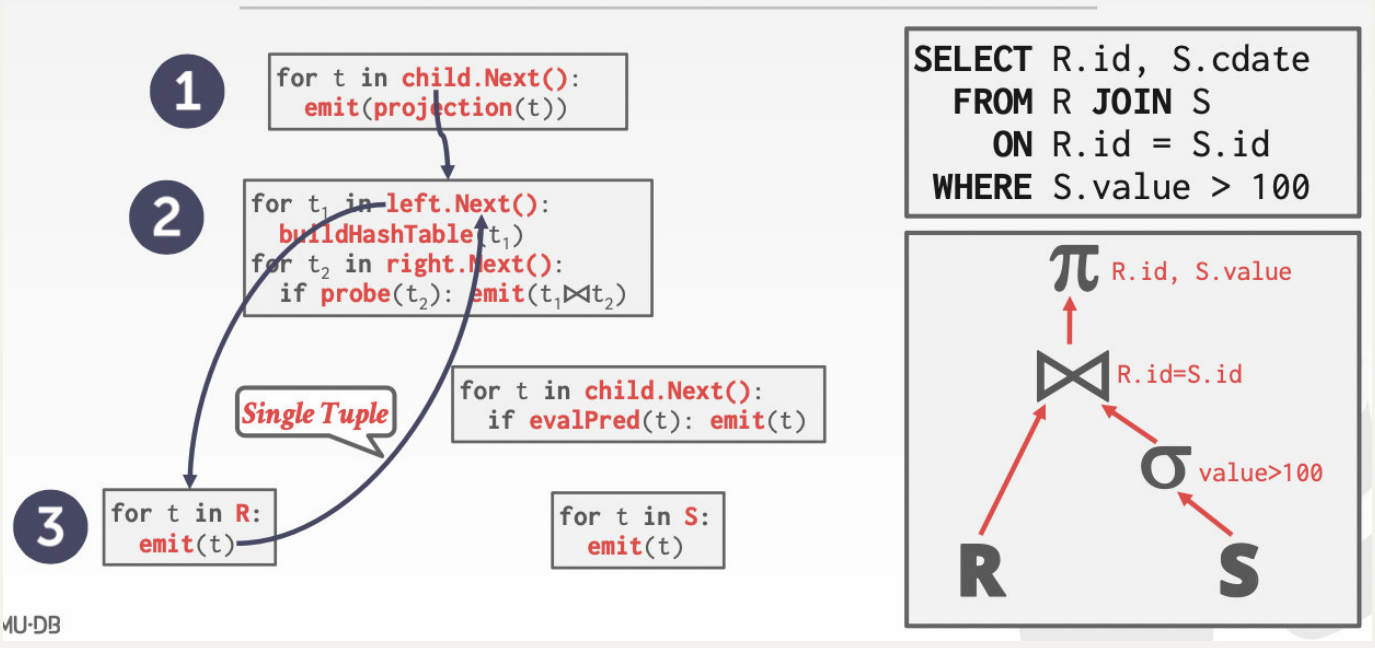
Iterator Model 迭代器模型/火山模型/流式模型

每个算子/操作符要实现一个`Next()`方法，每次调用它时操作符会返回一个tuple或者null，null表示tuple已经都被返回完了，操作符本身由一个循环实现，循环内部调用其子操作符的`Next()`方法，并从它们那边获取下一条数据供自己操作

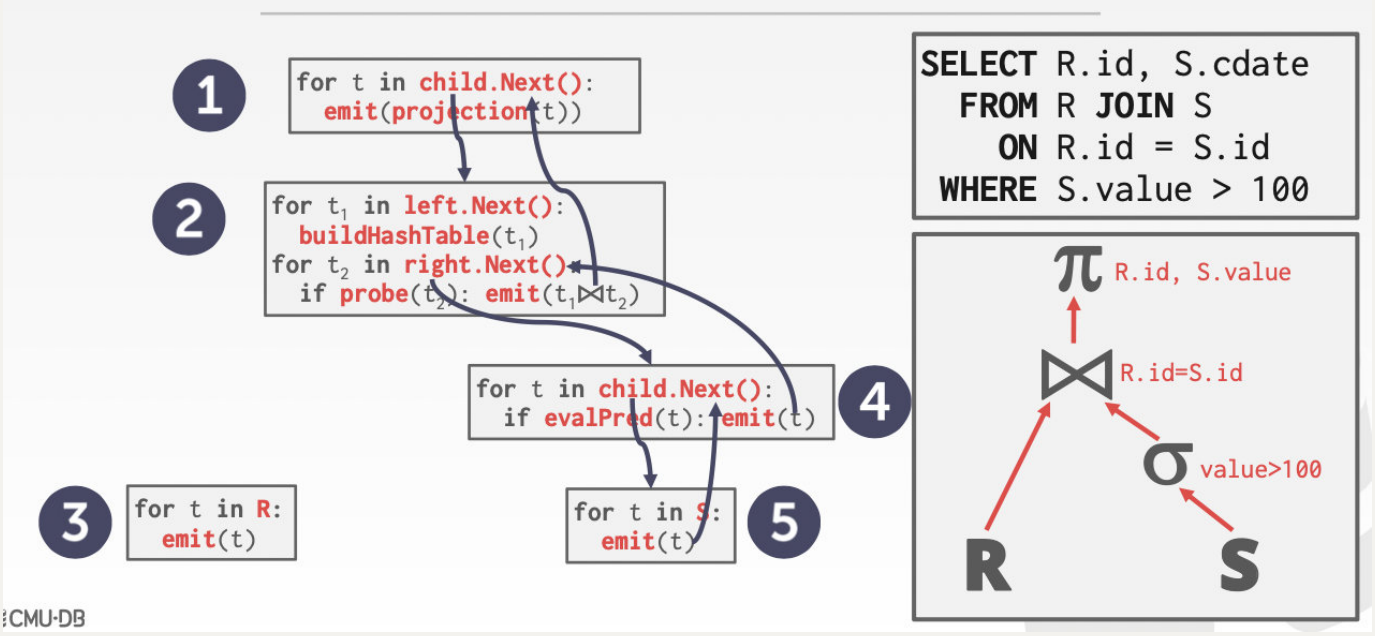




结合下图场景分析，位于根节点的投影算子的实现方式就是循环地调用其子节点的`Next()`方法，然后将所有返回的tuple经过投影处理后输出。其子节点join算子的`Next()`函数的实现方法是（其实就是上个Lecture中的各种join算法里的hash join）：先循环地调用其左子节点的`Next()`函数，用所有返回的tuple（它们合起来就是outer table）去构建哈希表（此时等待join算子的`Next`函数返回的投影算子处于阻塞状态）



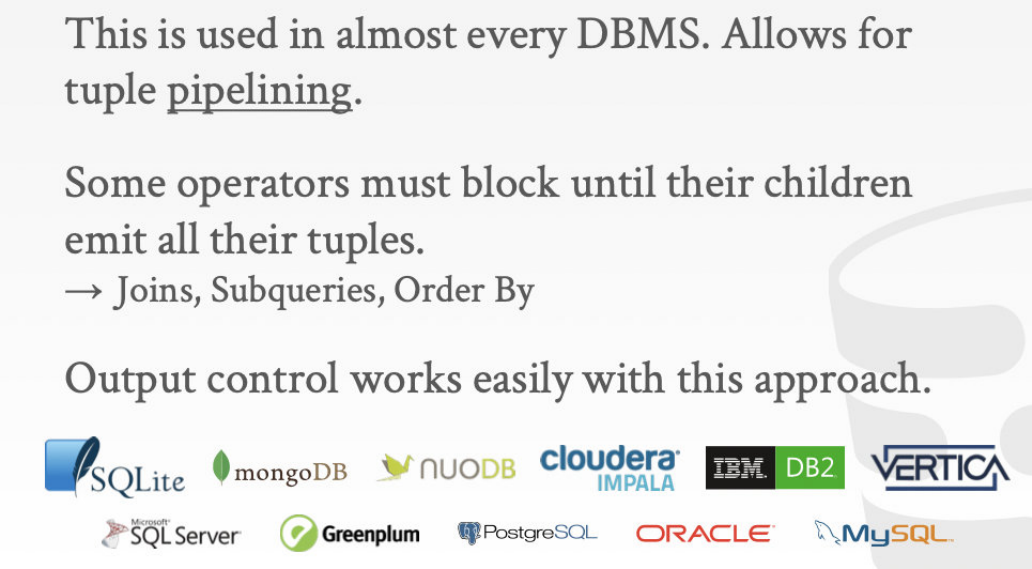
之后循环地调用其右子节点（筛选算子）的`Next`方法，每次调用时，右子节点吐出一个tuple，然后拿着join算子拿着这个tuple的join key去刚刚构建的哈希表里查询，查看能否成功匹配，如果可以的话，那就返回一条join后的结果，即向其父算子（投影算子）吐一条数据



这种处理模型执行SQL语句的方式，如同岩浆从火山底部往火山口汇聚，最终喷发，因此得名“火山模型”

火山模型本质上和基于栈帧的过程调用有关，上图中的箭头其实也指明了调用链，即调用关系/栈上有哪些函数的栈帧

几乎所有的DBMS都会使用火山模型或者是它的变种，并且火山模型有一些“流处理”（笔者认为这和计算机体系结构中流水线类似，都可以做到让系统的每个组件在同一时间都有事情做）的思想在内，上图中join算子的右子树中，S表的一个tuple被通过`Next`函数吐出后传递给了筛选算子，筛选算子在经过筛选后再把它传递给join算子，join算子经过join操作后再把它传递给投影算子。这个过程是一条一条数据向上流动的，这和我们直觉上想象的查询语句执行模式是不同的，直观上我们一般认为是S表整个被传递给了筛选算子，筛选算子把S表筛完了之后，把整个结果一次性地传给join算子，join算子处理完了整个的inner table之后再把join的结果一次性地传递给投影算子

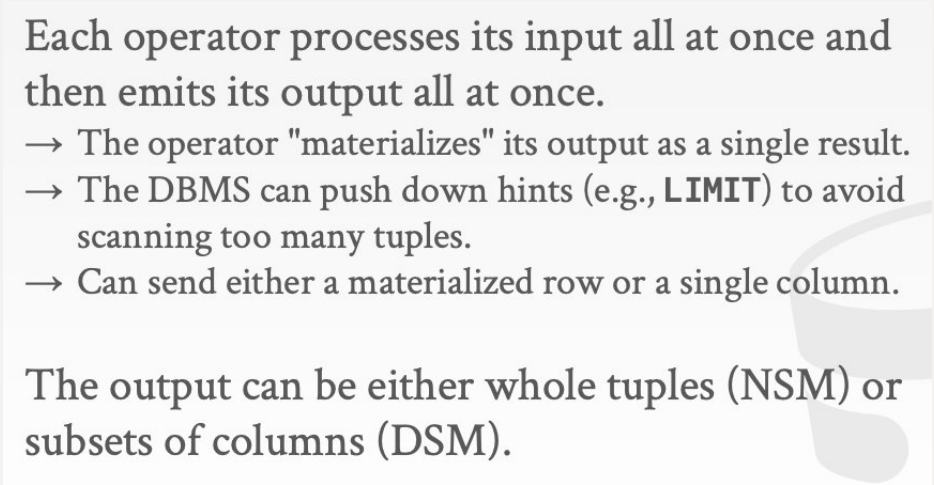


Order By操作符需要等到子操作符把所有tuple通过`Next`函数返回之后进行排序，最后按顺序输出，因此在这个过程当中它和它的父操作符都处于阻塞状态

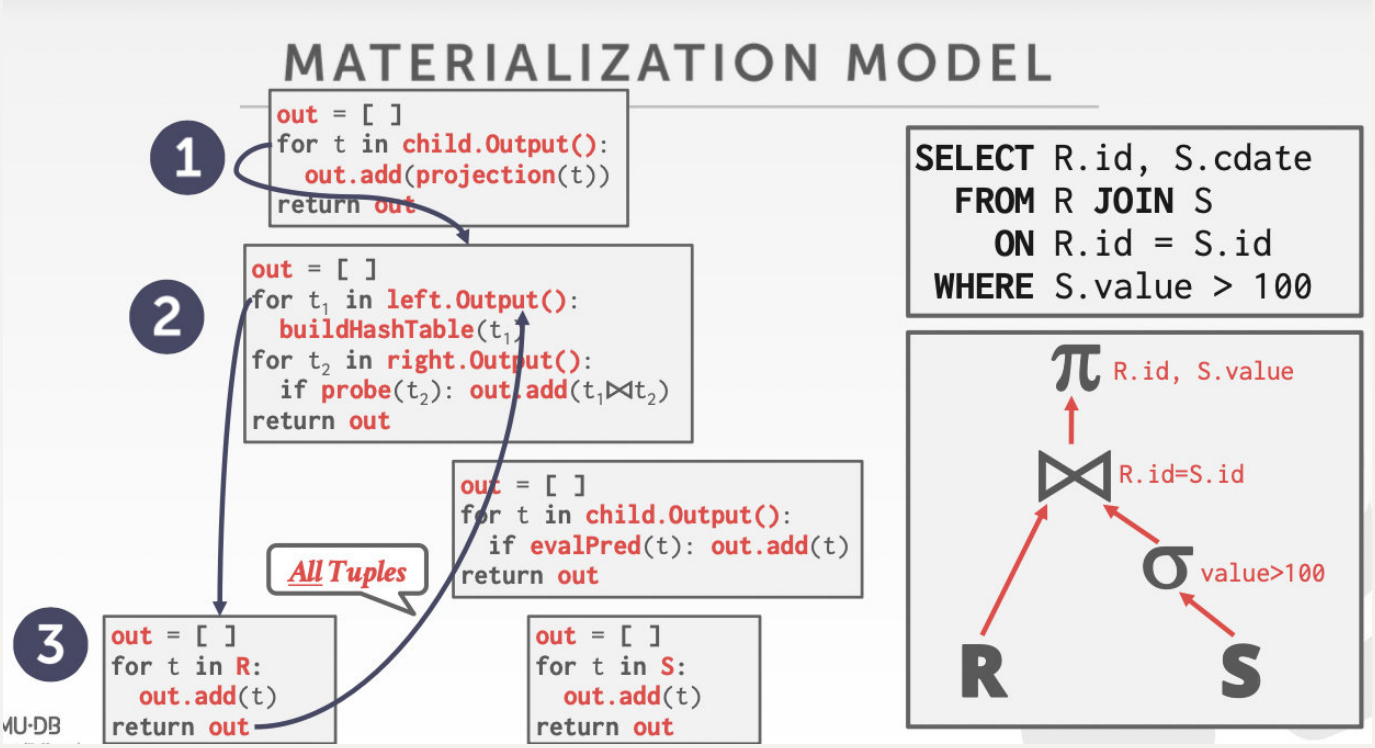
火山模型便于实现对输出的控制，比如说SQL语句中有"limit 100"这样的关键字，限制只输出100条数据，在火山模型下我们只需先流式地输出100条，然后让顶端的算子停止输出。我们不需要额外控制最底层的table reader（也就是读表的算子），只需要在操作符树的顶端控制数据的出口。但火山模型在性能上也有一些问题，每一条数据的传输都依赖函数调用，虽然函数调用的开销远小于硬盘IO，但如果要上千万条这样大量的数据向上流动，函数调用的次数将非常多，这会降低性能

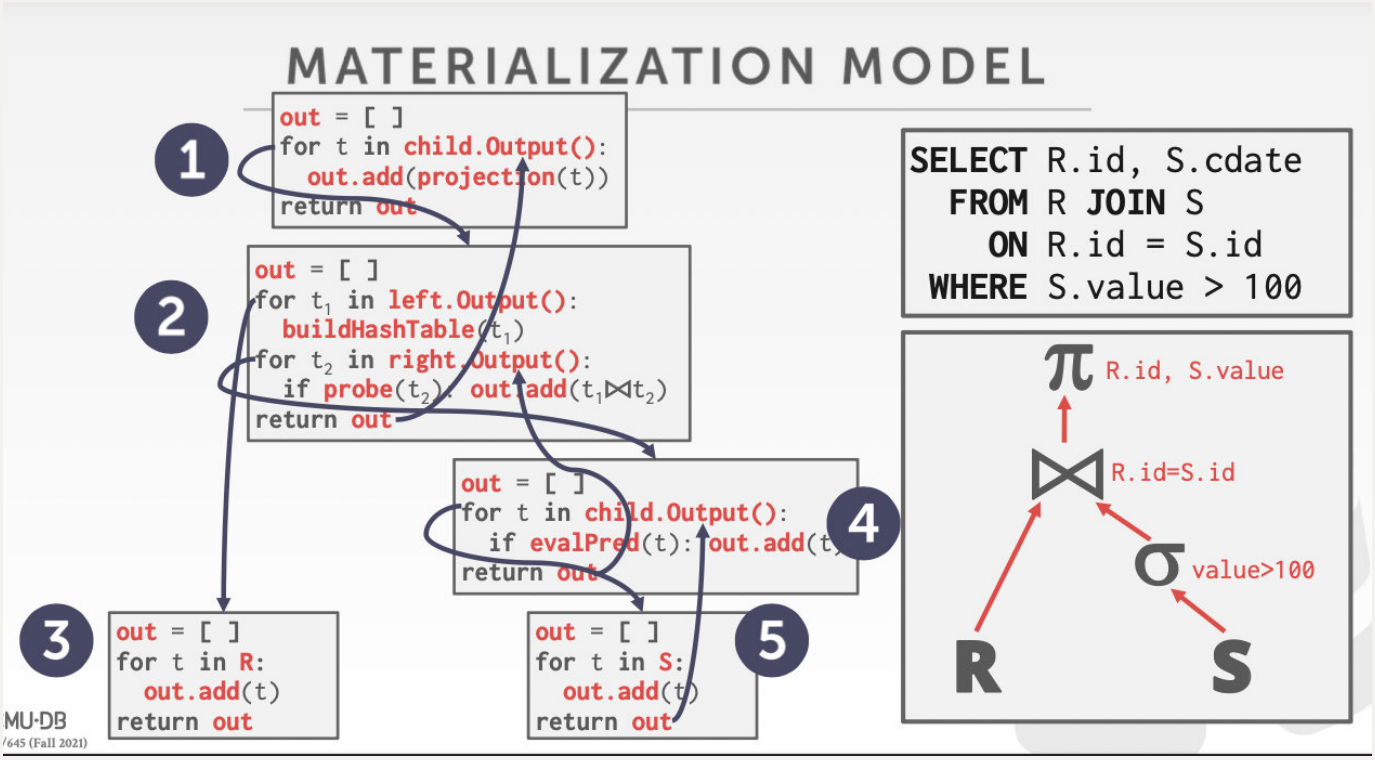
Materialization Model 物化模型

物化模型指的就是前面所说的“直觉下的查询语句执行模式”：算子一次性读入全部要处理的数据，将得到的结果一次性地输出。



如下面场景所示，每个算子对应的函数的返回值是数组形式的，一次性地将所有结果返回，并且会通过调用它的子算子对应的函数来一次性地获取要处理的数据

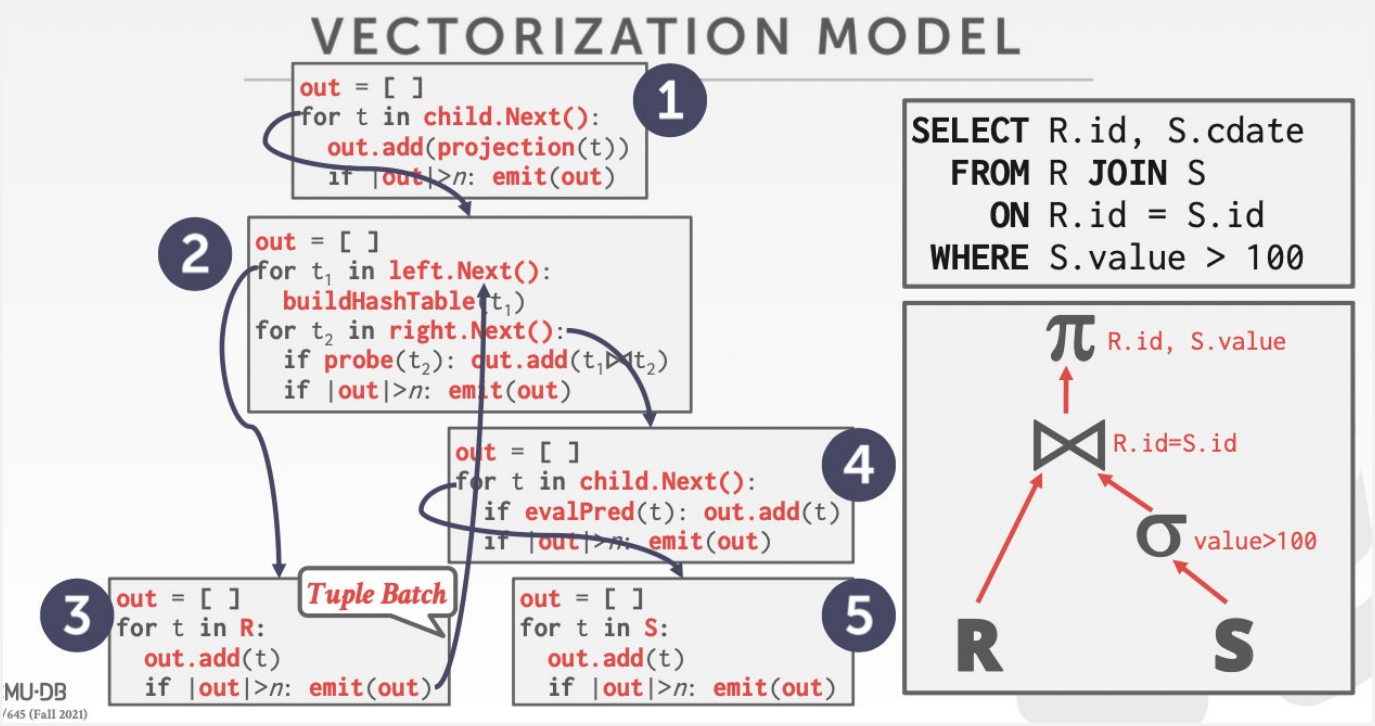




交易和事务对应的OLTP型数据库会使用这种模型，因为OLTP通常是点查询，只会涉及几条数据，每次子算子吐给父算子的数据不多，DBMS可以接受，如果是OLAP的负载，那么每次函数调用会返回过多的数据，DBMS无法承受

Vectorized/Batch Model 向量化模型/分批模型

火山模型每获取一条数据就要经过一系列的函数调用，物化模型每次函数调用可以获取很多数据，它们有各自的优点和缺点，向量化模型是这二者中和的产物，向量化模型中，每个算子也有`Next`函数，但它返回的不是一条数据，而是一批数据（tuple batch），这样可以减少函数调用的次数，从而降低开销



这种模型对经常进行大数据分析的OLAP型数据库比较友好，既能做到向上层算子返回的数据量不是太大，又可以控制函数调用的次数与开销。如果CPU支持SIMD指令集（比如说Intel的AVX指令集，这是CISC指令集所独有的一种指令），SIMD指令集又名向量指令，可以将一组操作数送上CPU，然后一次性把这一组操作数要进行的运算同时完成（前提是这一组操作数要执行的运算是同一种，比如说都做加法），那么这将更适合向量化模型，因为向量化模型中子算子给父算子每次传递的是一批数据，这一批数据要进行同样的运算（比如说传送给筛选算子的话，执行的全是比较大小之类的运算），一个SIMD指令就能让负责接收数据的父算子一次性完成对这一批数据的操作，这从硬件层面上大大地加速了DBMS查询语句的执行

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

Allows for operators to more easily use vectorized (SIMD) instructions to process batches of tuples.

DBMS计划执行/函数调用的方向分为两种，一种是让父算子调用子算子，自顶向下，另一种是子算子完成操作之后调用父算子，自底向上，但不论如何，数据流的方向始终是从操作符树的叶子节点流向根节点

Approach #1: Top-to-Bottom

→ Start with the root and "pull" data up from its children.

→ Tuples are always passed with function calls.

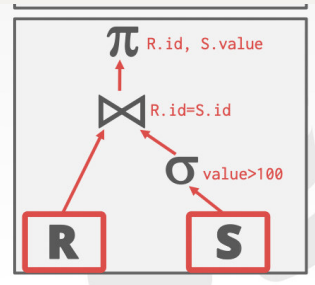
Approach #2: Bottom-to-Top

→ Start with leaf nodes and push data to their parents.

→ Allows for tighter control of caches/registers in pipelines.

Access Methods

这个部分讨论的是DBMS访问表的办法（也就是相关的算子读下图中的R表和S表的方法），



有如下三种基本方式

Sequential Scan 顺序扫描

简单地说就是一页一页地扫描，把硬盘里的每一页读入内存之后就一条一条地开始扫描其中的数据，之后"do something"

For each page in the table:
→ Retrieve it from the buffer pool.
→ Iterate over each tuple and check whether to include it.

```
for page in table.pages:
  for t in page.tuples:
    if evalPred(t):
      // Do Something!
```

The DBMS maintains an internal **cursor** that tracks the last page / slot it examined.

全表遍历式的顺序扫描往往是查询计划执行时的性能瓶颈所在，因此顺序扫描也有一些相关的优化策略

SEQUENTIAL SCAN: OPTIMIZATIONS

This is almost always the worst thing that the DBMS can do to execute a query.

Sequential Scan Optimizations:
→ Prefetching
→ Buffer Pool Bypass
→ Parallelization
→ Heap Clustering
→ Zone Maps
→ Late Materialization

Prefetching是说如果缓存池知道接下来要进行顺序扫描的话，它会在执行器尝试读数据之前把要读的数据预加载进入缓存池；

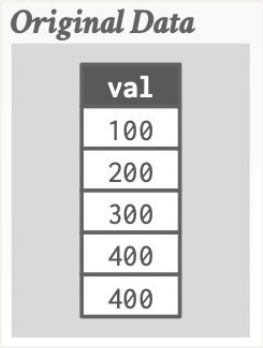
Buffer Pool Bypass是说在顺序扫描的时候不把当前扫描的页送入缓存池，而是在内存中另外使用一块区域，当执行器扫描完这块区域里的数据之后就把这块内存释放，这样做的好处是：顺序扫描时被扫描过的页以后会有很大概率不会再访问，不把它放入缓存池的话可以让其他需要被缓存的页继续呆在缓存池里，而不是过一段时间后因LRU策略被踢出

Parallelization是说我们可以让多个需要做全表顺序扫描的算子并行地以多线程的形式去访问表，比如说thread 1从头开始读，thread 2从中间开始读，etc.

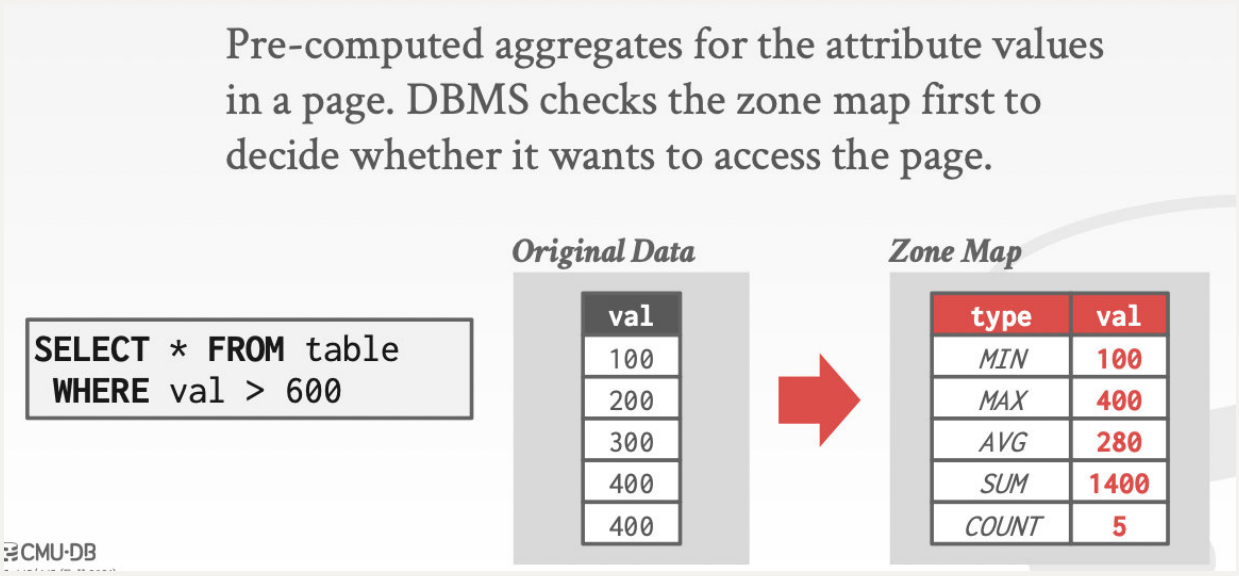
上图中的后三个策略将在下面着重介绍：

- Zone Maps

  首先假设有如下的场景，某个线程想扫描全表找到val字段的值小于100的tuple，但当前被扫描的页里没有符合要求的tuple，那么对这个页进行的扫描就是无意义的，只会徒增开销



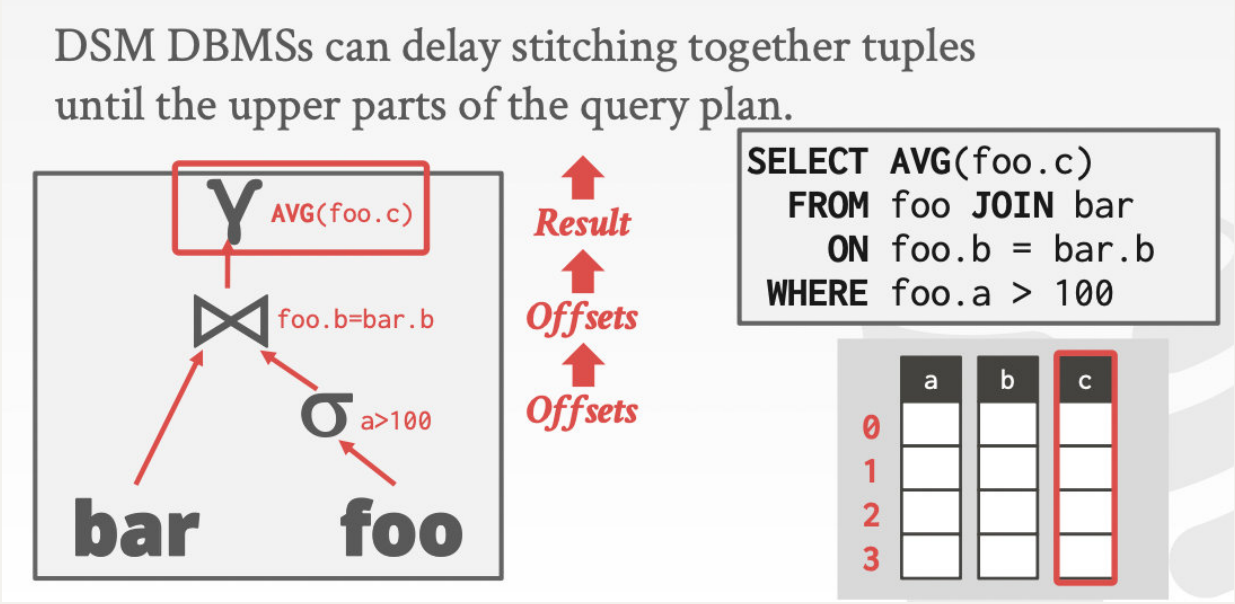
  为了解决这种问题，我们不妨给表的每个页做统计信息（如下所示），统计所感兴趣的字段的最小值/最大值/平均值/总和/数目，将这样的metadata存放在硬盘的其他位置（不是数据页中）。那么比如说当需要读表的操作符（table reader）需要读表然后选择符合条件的tuple时，可以先看每个页的统计信息，如果通过统计信息能推断出该页里没有我们想要的数据，那我们就不用把这个页读入缓存池里，然后直接去尝试访问表的下一个页，这便提升了顺序扫描的效率



  但这么做也是有一些缺点的，会导致额外占用存储空间存放Zone Map这样的metadata，并且在当表被修改了之后，相关页的统计信息也要随之改变，因此就会有维护metadata带来的开销

- Late Materialization

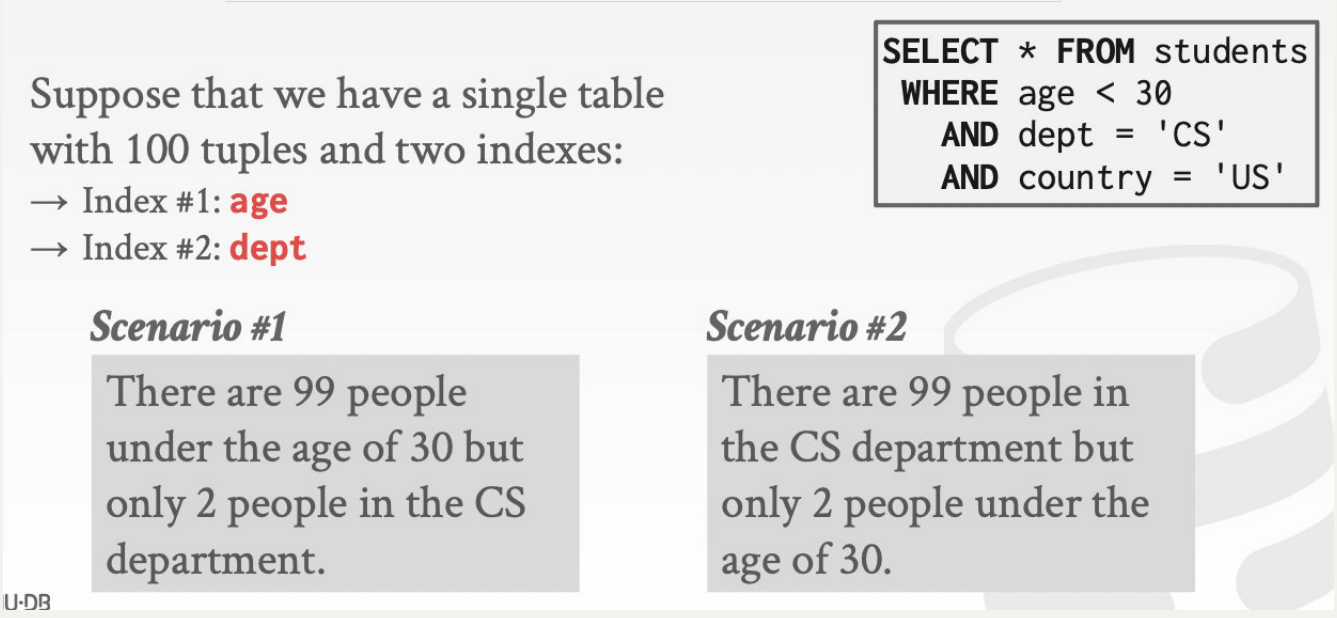
  存储引擎为列存储的DBMS中，算子输出的数据可以不是整条tuple，而是tuple在页里面的offset，这样就延迟了列存储情况下将不同列的字段拼接成完整的tuple的操作的发生，提升了效率



Index Scan 索引扫描

在上面的例子里面，我们想筛选出val字段的值大于100的tuple，其实可以尝试构建索引而非暴力地全表扫描，这样明显更有效率

如果我们有多个谓词（筛选条件），那么我们该先使用哪个谓词相应的字段的索引呢？原则就是“使用了某个索引之后，经筛选剩下的数据越少，就先使用这个索引”，详见下图场景：

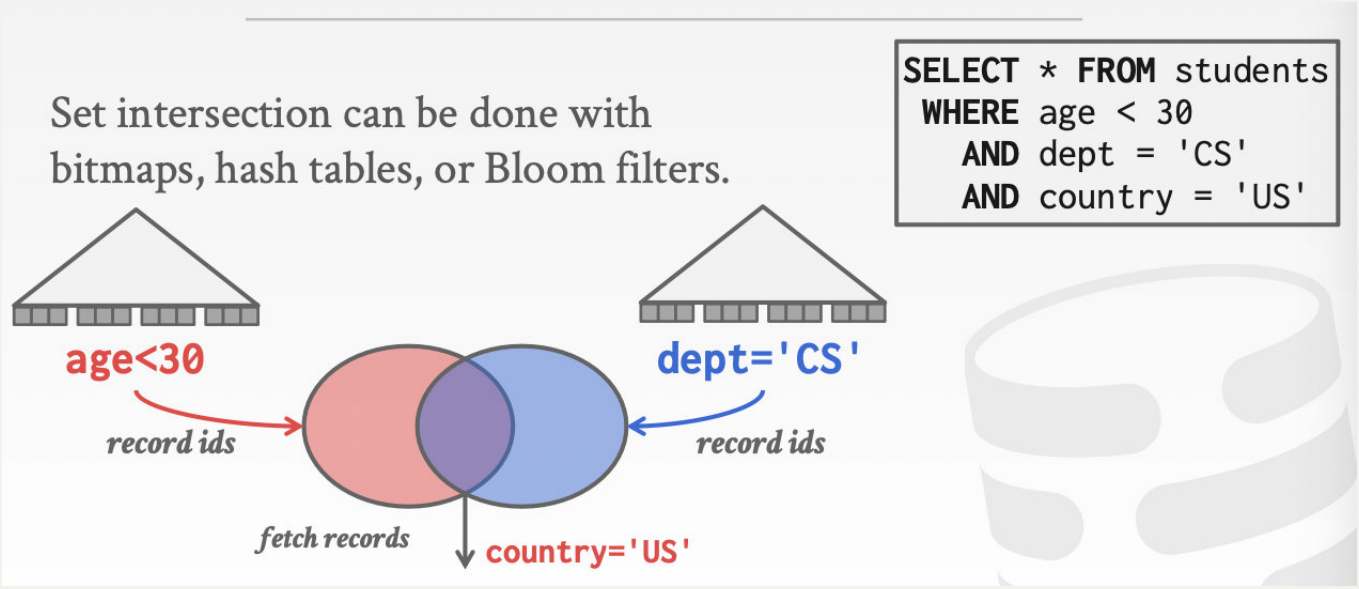


具体的索引的选择有很多的策略与trade-off，在Lec13查询优化那一节会涉及到

Multi-Index/"Bitmap" Scan 多索引扫描

还是上面的例子，我们也可以用age字段的索引把age<30的人筛选出来，同时也用dept字段的索引把dept='CS'的人筛选出来，之后取它们的交集，再从交集里面选取country='US'的人，取交集的过程可以通过bitmap/布隆过滤器等实现

With an index on **age** and an index on **dept**:
→ We can retrieve the Record IDs satisfying **age<30** using the first,
→ Then retrieve the Record IDs satisfying **dept='CS'** using the second,
→ Take their intersection
→ Retrieve records and check **country='US'**.



Modification Queries

前面讲的都是读表的方法，属于`SELECT`语句的内部实现方式，但还有好多其他种类的SQL查询语句，诸如`INSERT`,`UPDATE`,`DELETE`这些，它们会修改表的内容，它们的执行逻辑与`SELECT`完全不同，它们需要检查约束（e.g. 如果不允许表的某一列存在重复的元素的话，就不可以向表中随便插入数据），维护索引（e.g. 插入新的数据之后也要更新索引）

Operators that modify the database (**INSERT**, **UPDATE**, **DELETE**) are responsible for checking constraints and updating indexes.

UPDATE/DELETE:

→ Child operators pass Record IDs for target tuples.

→ Must keep track of previously seen tuples.

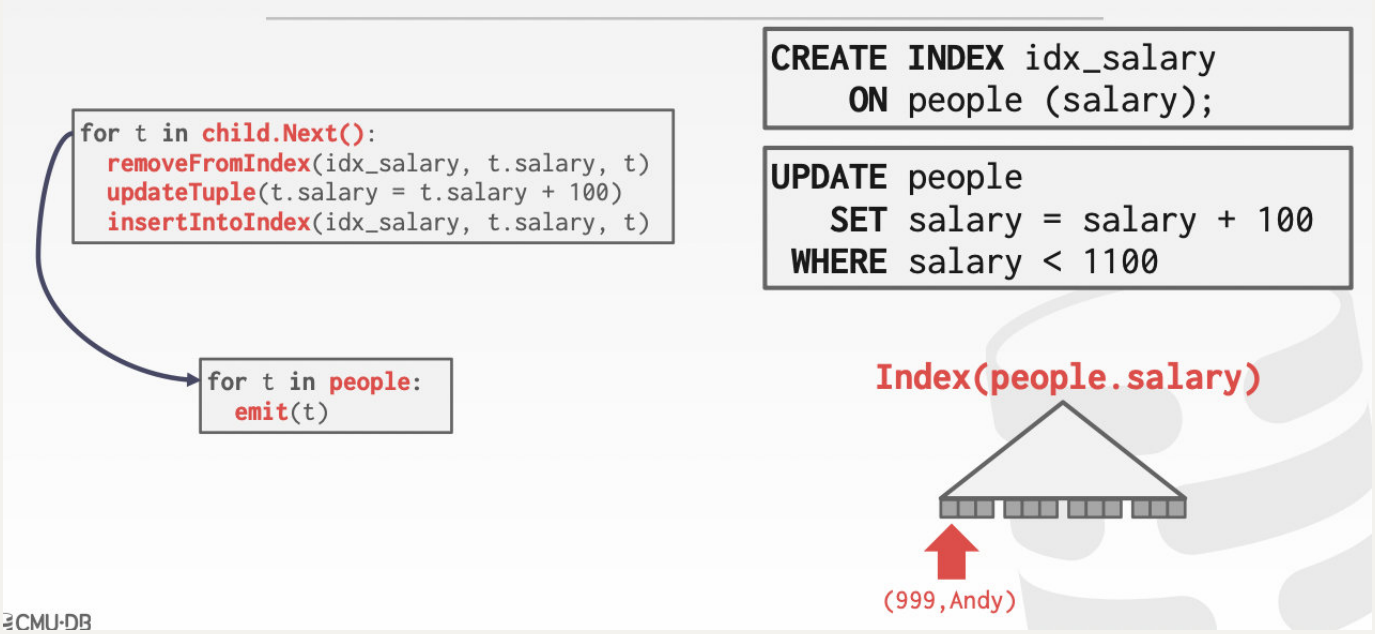
INSERT:

→ **Choice #1**: Materialize tuples inside of the operator.

→ **Choice #2**: Operator inserts any tuple passed in from child operators.

`UPDATE/DELTE`查询语句执行时，子算子会把要处理的tuple的id传递给上层负责完成更新/删除操作的父算子，然后父算子通过id找到相应的tuple，然后执行对应的操作。

此外，负责更新/删除的算子必须要记住在执行本次的查询语句时操作了哪些数据。这句话有些抽象，具体的意思还需结合如下的具体场景分析：名为people的表有关于salary字段的索引，我们在数据库中要给所有salary小于1100的人工资加100



执行SQL语句时，上图中上方负责完成更新操作的算子会先把和当前tuple有关的索引删除，然后更新tuple，最后将该tuple重新插回索引。因此，当扫描到salary为999的Andy对应的tuple时，会先从索引里删除该tuple对应的索引项，然后更新tuple，再重新插入对应的索引项，那么问题来了，新的tuple里面salary字段的值是1099，因此会被插到作为索引的B+树的后方的叶子节点里，而当前遍历叶子节点用的“光标”（cursor）还在Andy的tuple原先的位置。也就是说，光标继续往前挪动的时候，会再次碰到这个已经被改过tuple，而且由于它的salary字段是1099，小于1100，因此还会再加100，这就引起了错误。

所以说负责完成更新操作的算子应该记住它更新过了哪些数据

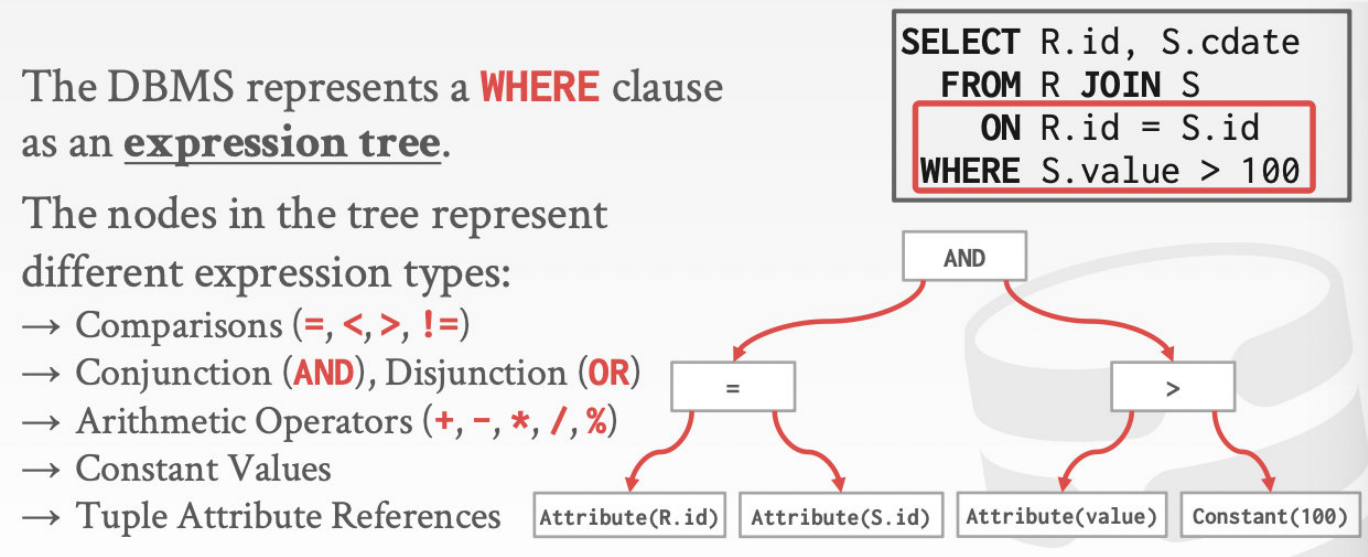
`INSERT`查询语句有两种执行方案，

- 在负责完成插入操作的算子内部物化tuple，之后把完整的tuple插入表中，并维护索引
- 需要子算子完成tuple的物化，然后传递给负责插入操作的算子，负责插入操作的算子只需完成最终的插入操作以及维护索引

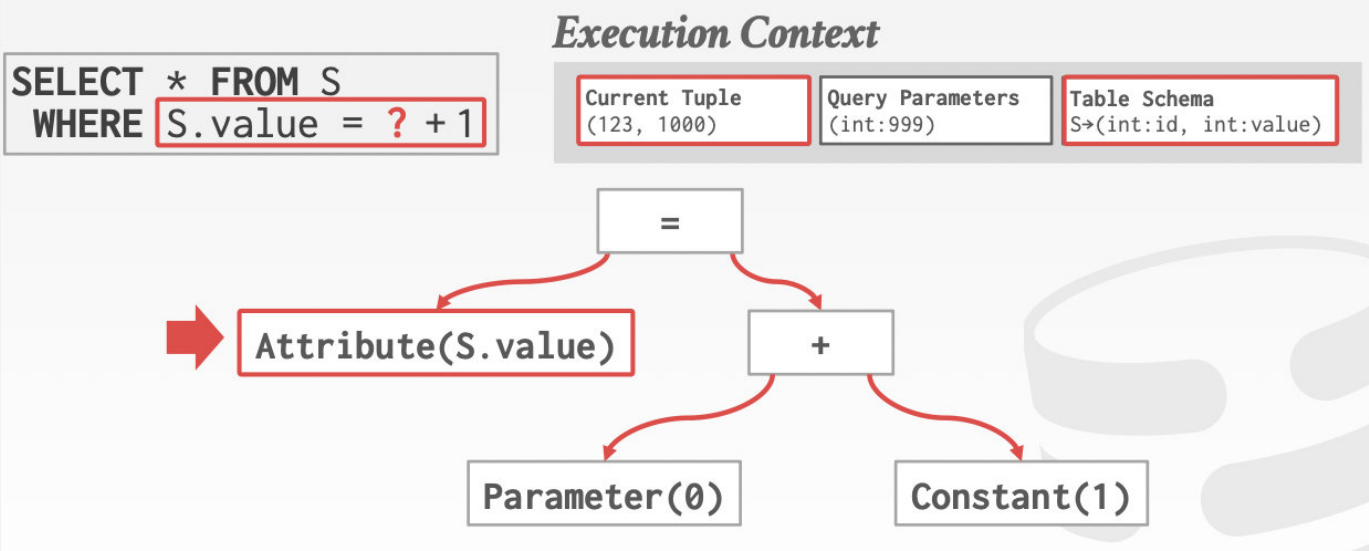
Expression Evaluation

最后介绍表达式计算的问题，SQL语句中的谓词本质上就是表达式

一个通用的方法是解析SQL语句时为其构建如下以算子为节点的树形结构

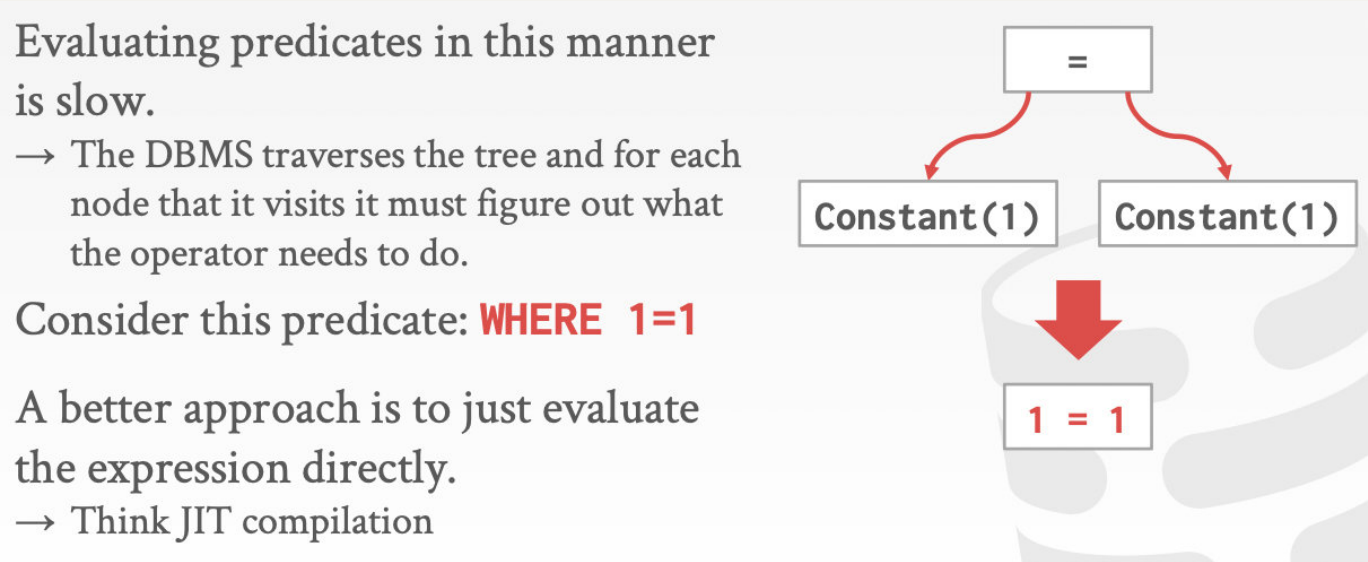


但是这个通用的方法在如下场景中会出现问题（SQL语句中?可以理解为待用户输入的参数）



每次获得S表的一个tuple的value字段的值后，都会计算用户传来的parameter和SQL语句里的constant的和，这会导致随着不断地扫描S表，这个加法被重复地进行了好多好多遍，这大大地降低了效率（和朴素斐波那契解法中时间复杂度爆炸是同一个原因）

这种情况下存在对应的优化策略：在执行SQL语句的查询计划前，先把语句中的常数提前算出来

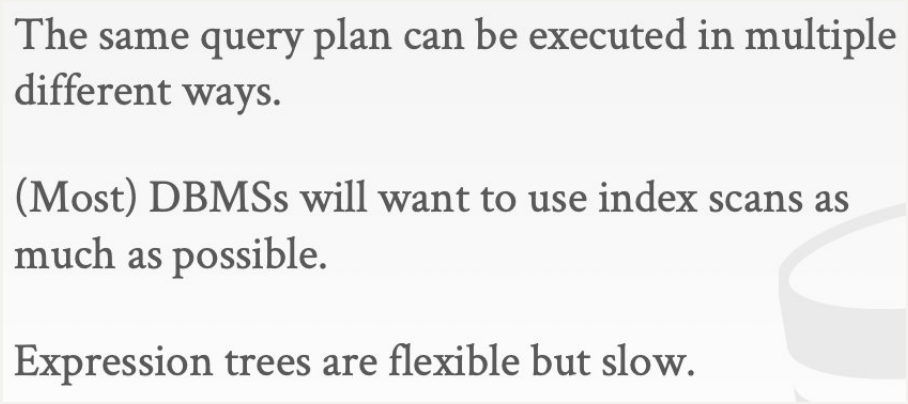


这和Java的高级技术JIT（即时编译技术）有相似之处，和JIT（Just in Time）对应的是AOT（Ahead of time）

在Java的JIT技术中，JVM虚拟机会将被频繁执行的热点代码段（比如说JVM能检测出来有些循环被执行了好多好多次，那这就可以被标记为热点代码段）中的字节码转化成二进制代码，下次再运行到这个热点代码段的时候就直接运行二进制代码，不经过中间那层虚拟机，从而提升了效率

JIT技术是在代码被执行的时候动态地判断出哪段代码是频繁被执行的，而AOT技术是在代码被执行之前就进行这样的判断

最后，本Lec的总结如下



判断DBMS的性能时，可以优先分析“DBMS内部创建了哪些索引”，“这些索引有没有被很好地利用”，这对DBMS的性能至关重要

Reference/参考自：

https://www.bilibili.com/video/BV1TR4y1W7ny/?spm_id_from=333.788

https://zhuanlan.zhihu.com/p/418937203