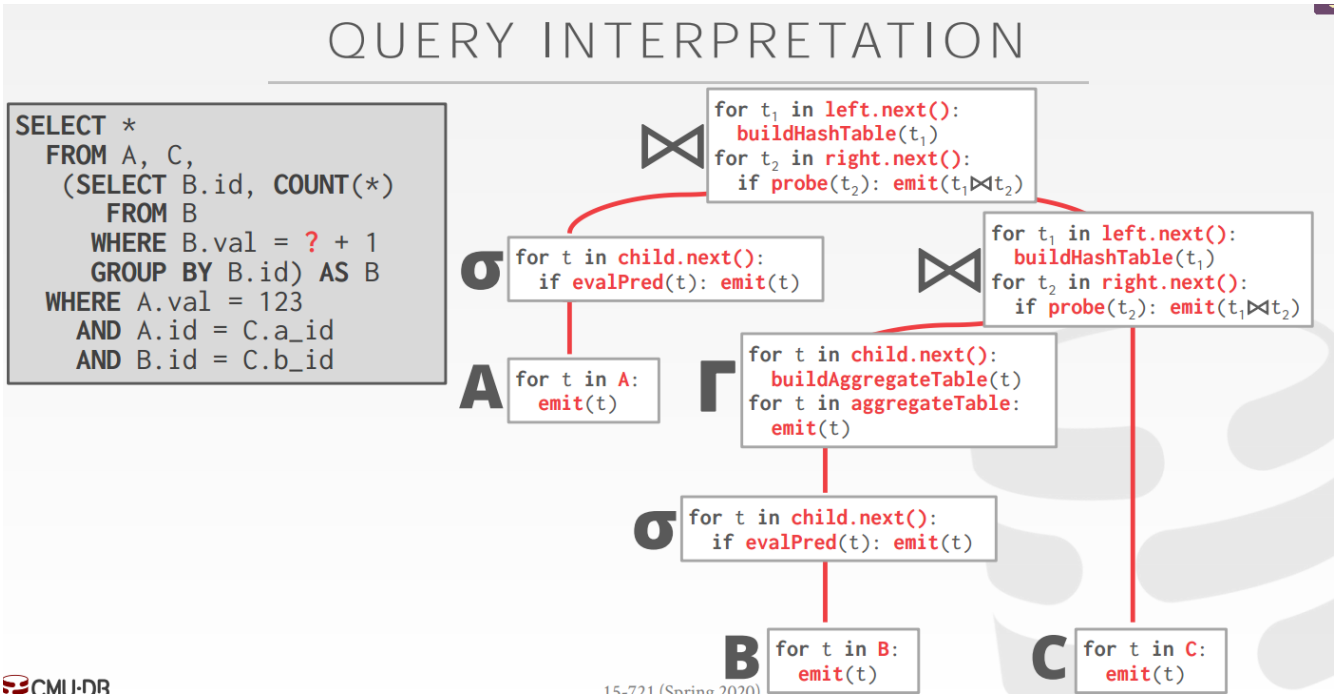
Query Compilation & Code Generation

在In-Memory DBMS中,没有了磁盘IO这一瓶颈。提高吞吐量的关键是**减少要执行的query数量**

一种方法叫**code specialization**,就是说针对特定的query(task)来生成特定的code，目的是尽可能消灭if,where clause,消灭数据类型转换等

举个例子:



上图是上节课讲的执行模型，如果每个operator都跑一个for loop,算子之间需要通过函数调用来传递数据。函数调用时非常expensive的。

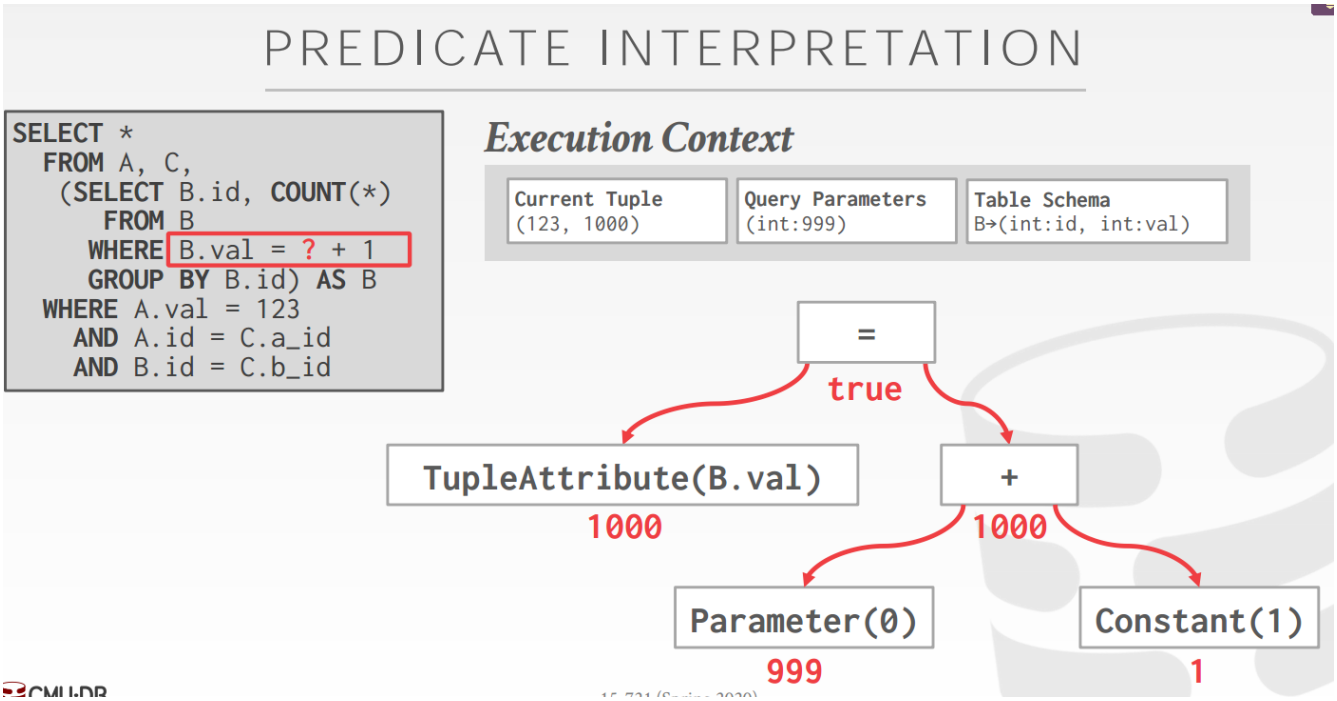
总结而言:

对于人类而言，这写代码以及这种查询树是非常好理解的。但是这段代码对CPU而言就非常不友好:

1. 过多的结构和分支：无论是for还是if都会产生大量的分支，导致CPU要不断刷新管道和缓存。
2. 大量的函数调用：函数调用导致CPU在内存中不断跳跃。

除此之外，谓词的判断过程也是非常expensive的。以`B.val = ? +1`来说，需要遍历的树结构如下

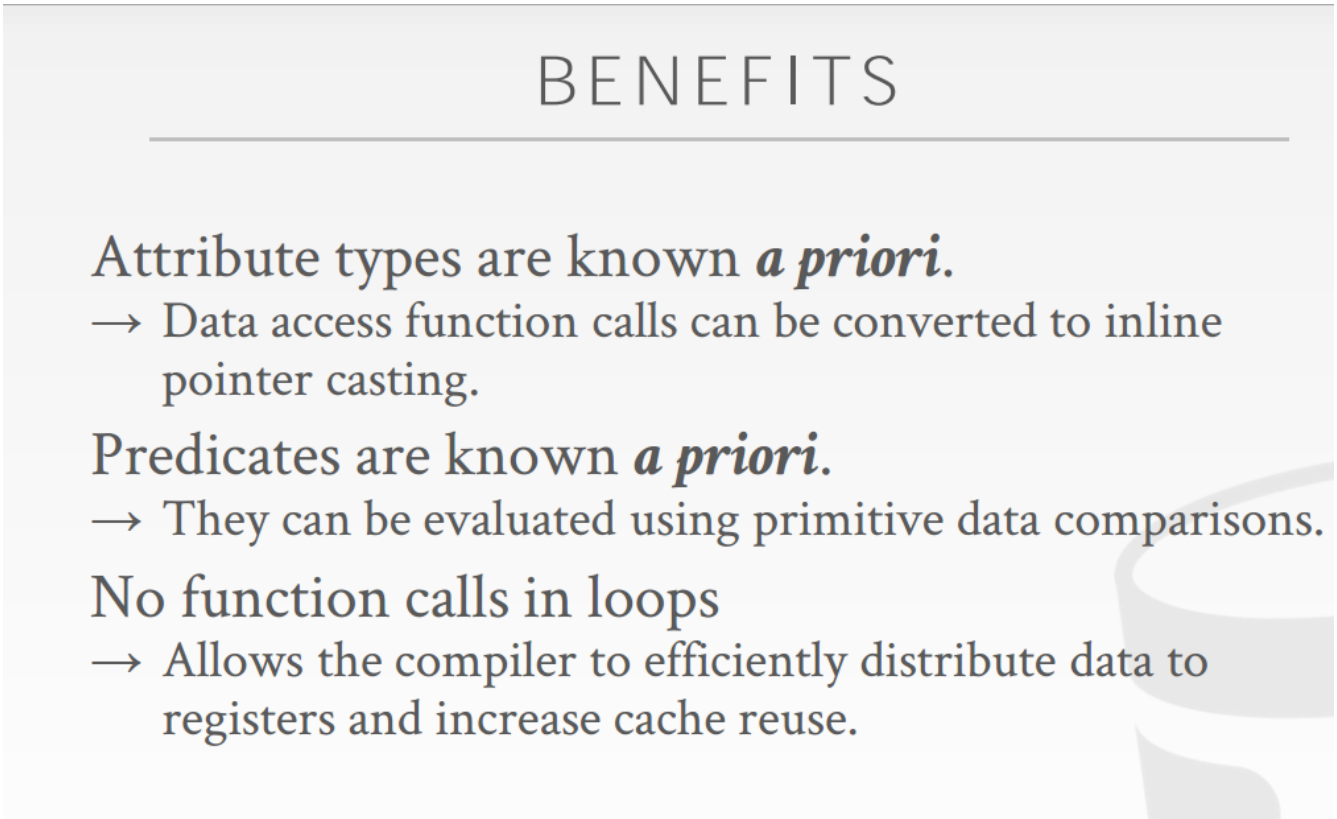
每一个节点都对应一次函数调用，可以看到仅仅为了处理一个元组就需要执行四次以上的系统调用。



所以我们才需要引入 **CODE SPECIALIZATION**

对不同的输入的相似执行模式的查询进行特定的编译设计。

这样好处是：



- 属性类型先验已知

访问数据的函数调用可以转换为内联指针转换。

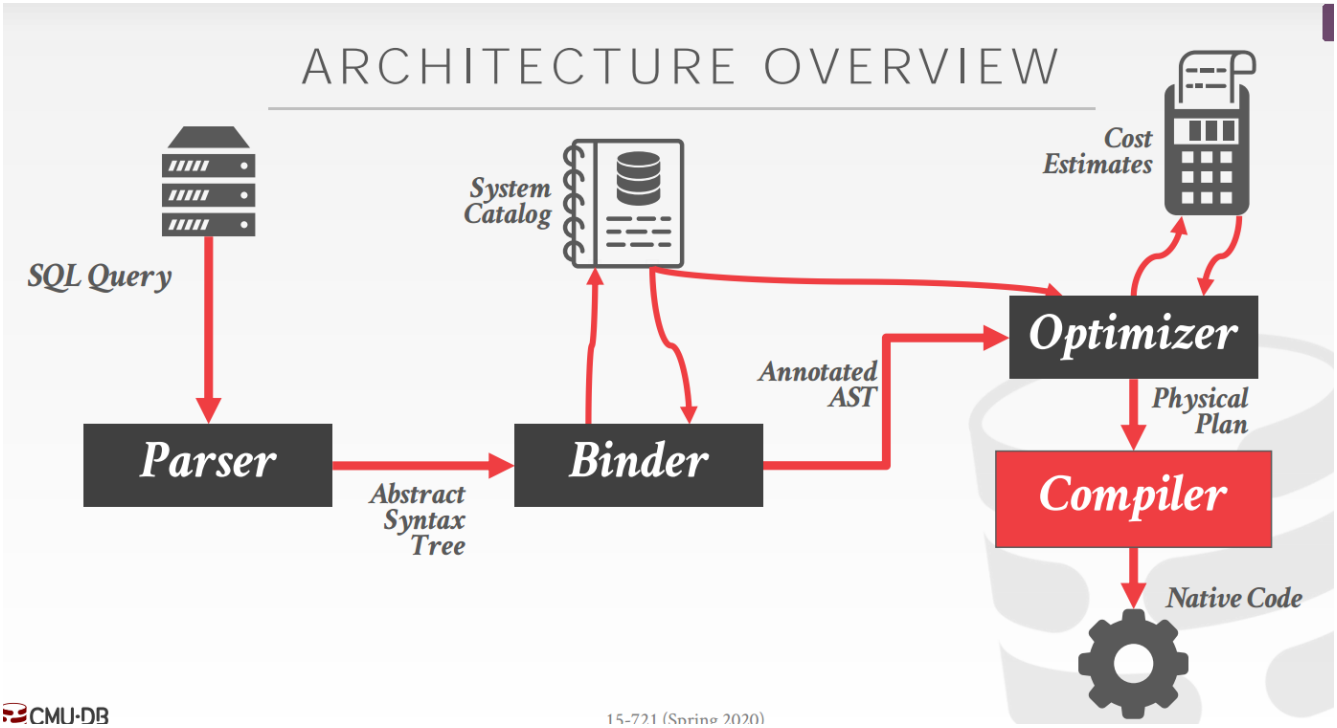
- 谓词先验已知

可以直接使用原始数据来比较

- 循环中没有函数调用

那么编译器可以直接把数据分发到寄存器提高缓存重用

SQL执行过程



- SQL QUERY经过词法语法语义分析生成抽象语法树
- Binder会在System Catalog(存储元数据的地方)做look up，把表的string token标志替换为语法树内部的idtenifiers，那之后再次寻找的时候速度就会快很多
- 标注后的语法树经由Optimizer生成物理计划
- 物理计划通过Compiler生成可供执行的字节码

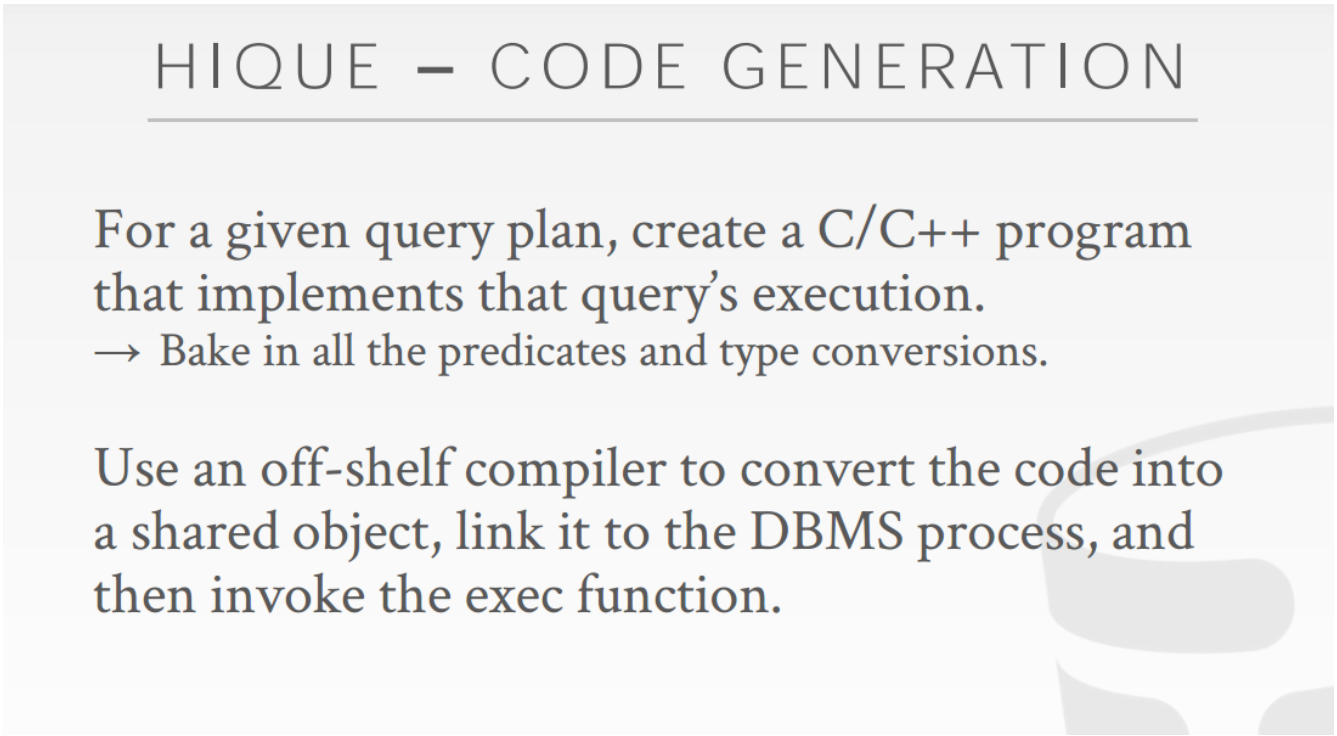
Approach1: Transpilation

编写将Query转换为C/C++的代码，然后通过常规编译器生成可执行机器码。

Approach2: JIT Compilation

即时编译，生成Query的中间表示。DBMS负责把中间表示转化为可执行的机器码。

HIQUE--FIRST DBMS TO DO CODE GENERATION



对于给定查询计划，创建一个实现该查询执行的C/C++程序，将所有谓词和类型转换都固定下来

然后编译器把代码转换成动态对象，连接到DBMS进程中，DBMS再调用exec来执行

直觉告诉我们调用exec系统调用还是非常损耗性能的。

生成的查询代码的组件可以调用DBMS中的任何其他函数。这允许它使用与Interpreted Plan使用相同的组件。

- 并发控制
- 记录/检查点
- 索引

HIQUE性能瓶颈主要在GCC编译所需要的时间。因为其需要查看环境，连接，库等。

JIT 即时编译(LLVM)

- 通过operator来组织查询时推断查询的有效方式，但不是执行查询最有效方法
- GCC编译速度其实很慢
- 不支持PIPELINing,，管道会可能因为其他数据没有处理完而等待其他数据的处理。

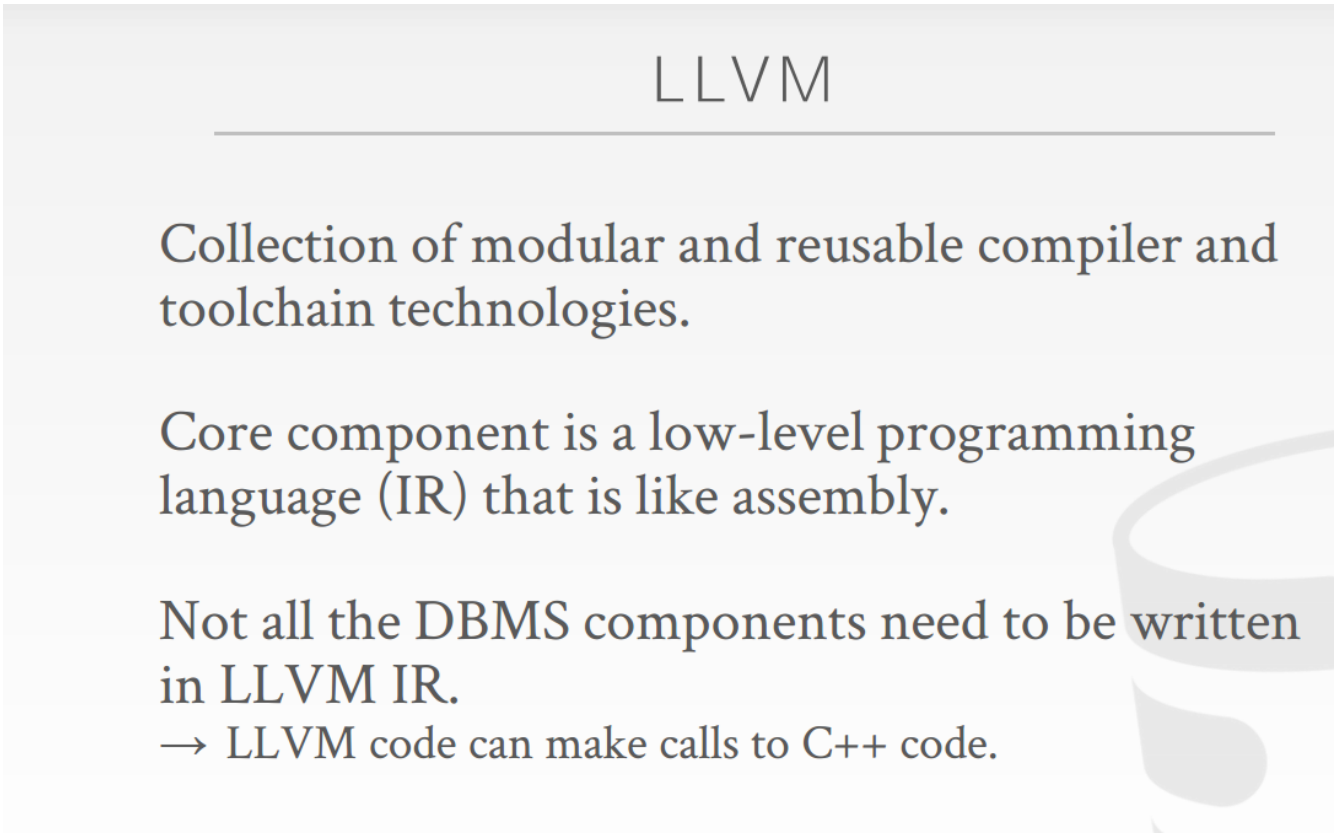
HYPER -- JIT QUERY COMPILATION

使用LLVM来在内存中对Query做即时编译。

目标是尽可能把tuple留存在CPU寄存器中。

- Push-based vs. Pull based
- Data Centric vs. Operator Centric

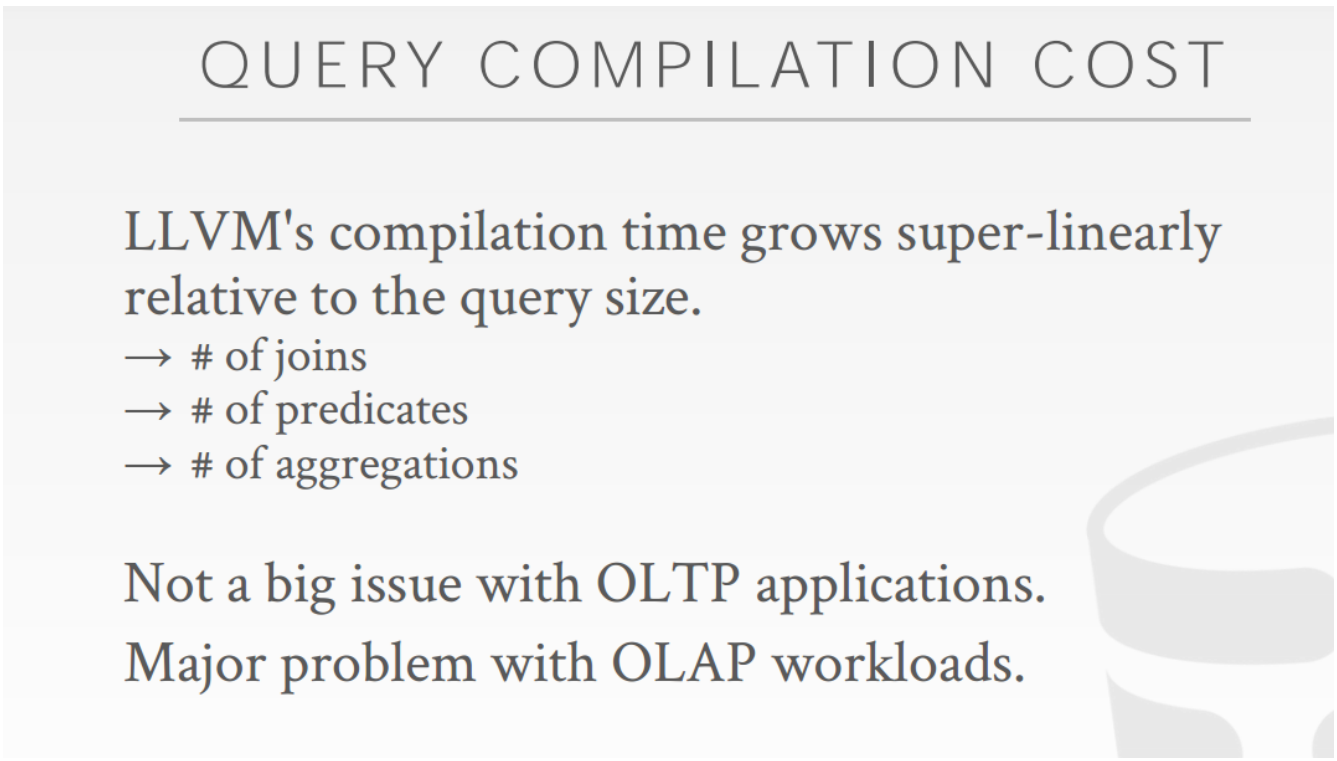
LLVM



但是LLVM的编译时间相比于query的执行时间来说还是太长了。

其实编译时间是与query的大小成线性关系。

如joins,predicates和aggregations的数量。



HYPER采取的办法是：结合解释器和即时编译生成的结果(和JVM的JIT原理相似)

First generate the LLVM IR for the query and then immediately start executing the IR using an interpreter.

Then the DBMS compiles the query in the background.

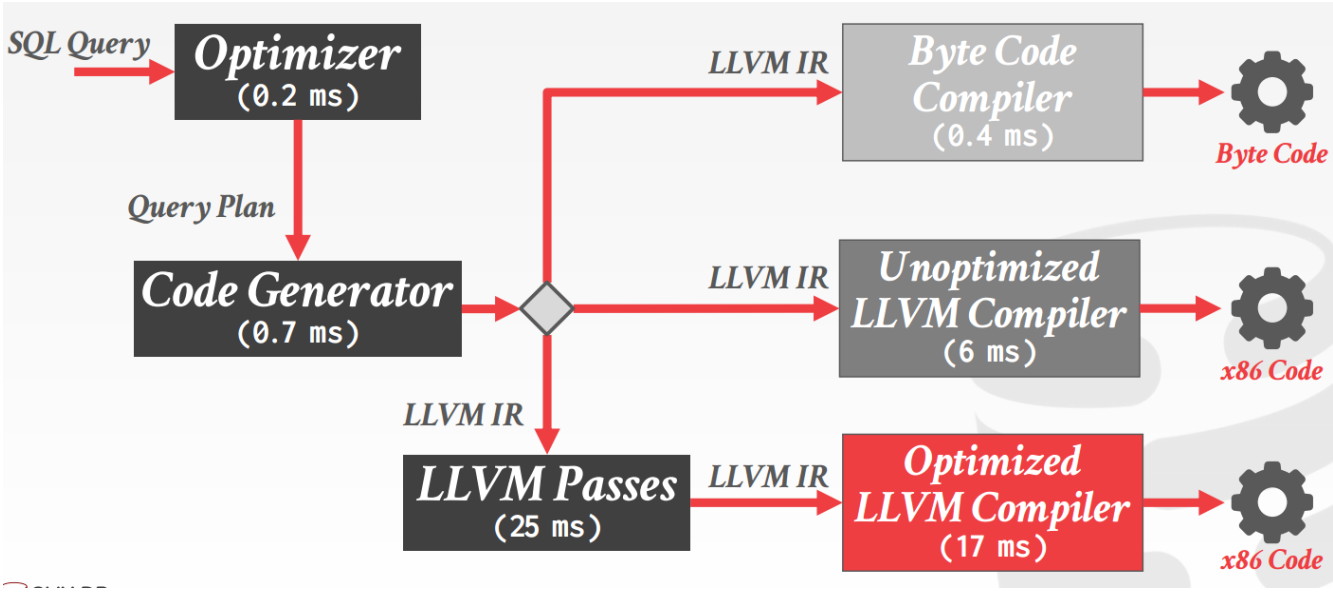
When the compiled query is ready, seamlessly replace the interpretive execution.

→ For each morsel, check to see whether the compiled version is available.

- 首先生成LLVM的中间表示，然后马上使用解释器来执行这些中间表示
- DBMS后台编译Query
- 由于解释器"解释"的本质其实就是把IR用已经生成好的预编译指令翻译一下，所以启动速度会比需要编译的编译器要快，但是编译之后的机器码执行速度会比解释器要快
- 因此编译完成之后，会把解释器解释的结果替换为编译后结果。

下图就是例子：

先是执行解释器生成的字节码，然后等LLVM Compiler编译完之后替换为执行编译后的x86Code



主流DBMS的即时编译实现：

REAL-WORLD IMPLEMENTATIONS

Custom	***JVM-based***	***LLVM-based***
IBM System R	Apache Spark	MemSQL
Oracle	Neo4j	VitesseDB
Microsoft Hekaton	Splice Machine	PostgreSQL (2018)
Actian Vector	Presto	Cloudera Impala
		Peloton
		CMU's DBMS 2.0