Paper Assigned: Runtime Code Generation in Cloudera Impala

Presentation by: Shivam Patel

What is Cloudera Impala?

- Impala is a MPP(Massively Parallel Processing) database for use in the Hadoop ecosystem.
- Uses HDFS and HBase(Hadoop version of Big Table) as storage managers.
- Uses Map Reduce for processing.
- Claims to execute queries in 10-100x speed compared to out of the box solutions.

How is the Speed up Achieved?

- (In part) By using run time information and JIT compilation to create optimized machine code that attempts to reduce instruction level latencies.
- Uses LLVM to compile and load "fully-optimized query specific" versions of functions at runtime for use in queries.

Instruction level latencies? Query Specific?

- For a particular query that uses a function, the function will typically act based on specific information that can only be known at runtime, hence when the function is written, it needs to handle general cases.
- When we write code to handle general cases, it involve excessive instructions to check things(types, number of elements, increment things)
- Example: Branch misprediction, Stalling over previous instruction reading/writing not being complete.

How does Impala(want to) solve this?

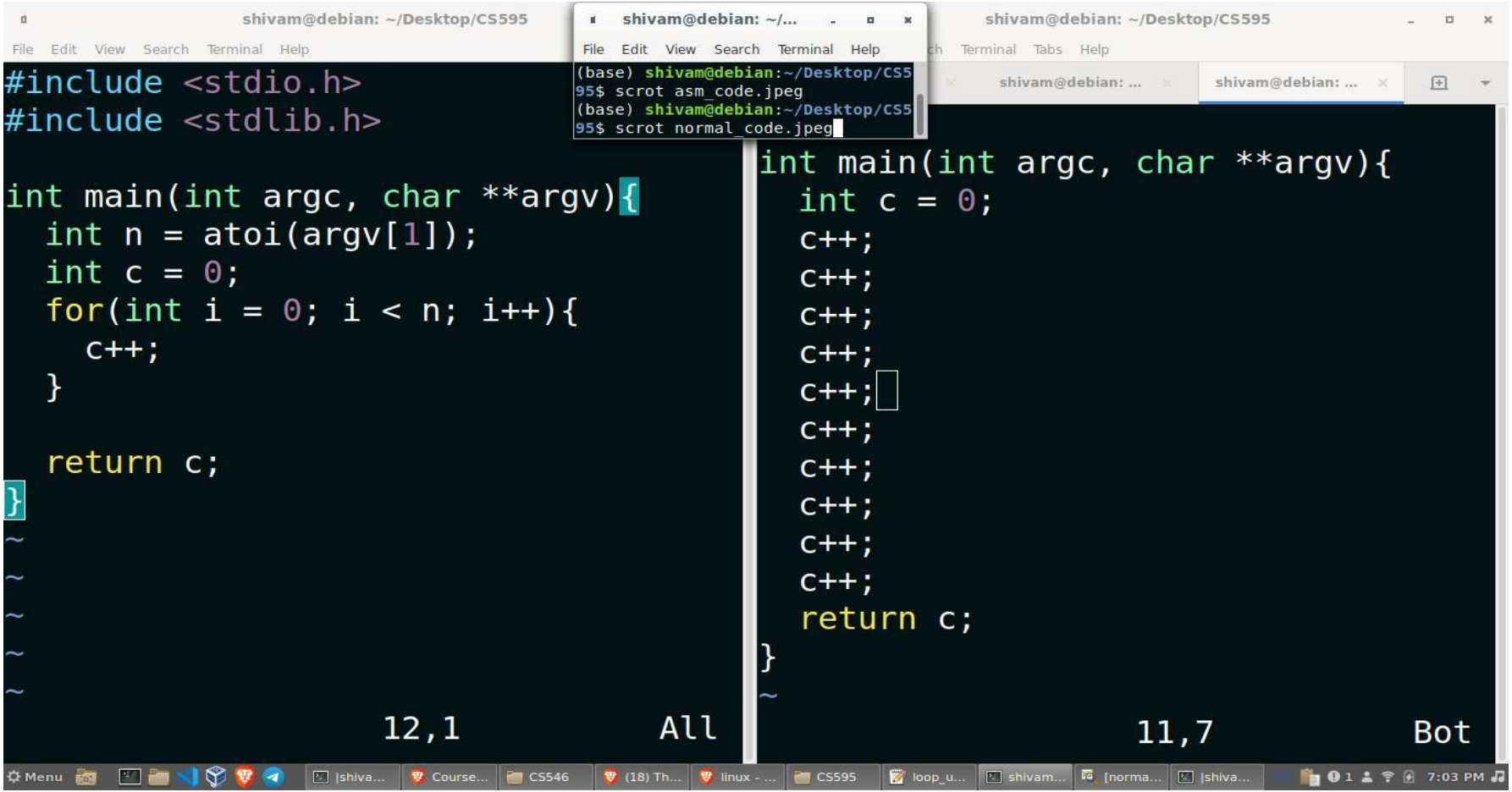
- When a query is about to be executed, invoke the JIT compiler LLVM offers to create specialized machine code using the runtime information and then use that instead of interpreting.
- Helps to:
    - Remove Conditionals
    - Remove Loads
    - Inline Virtual Functions

Q: What is gained?

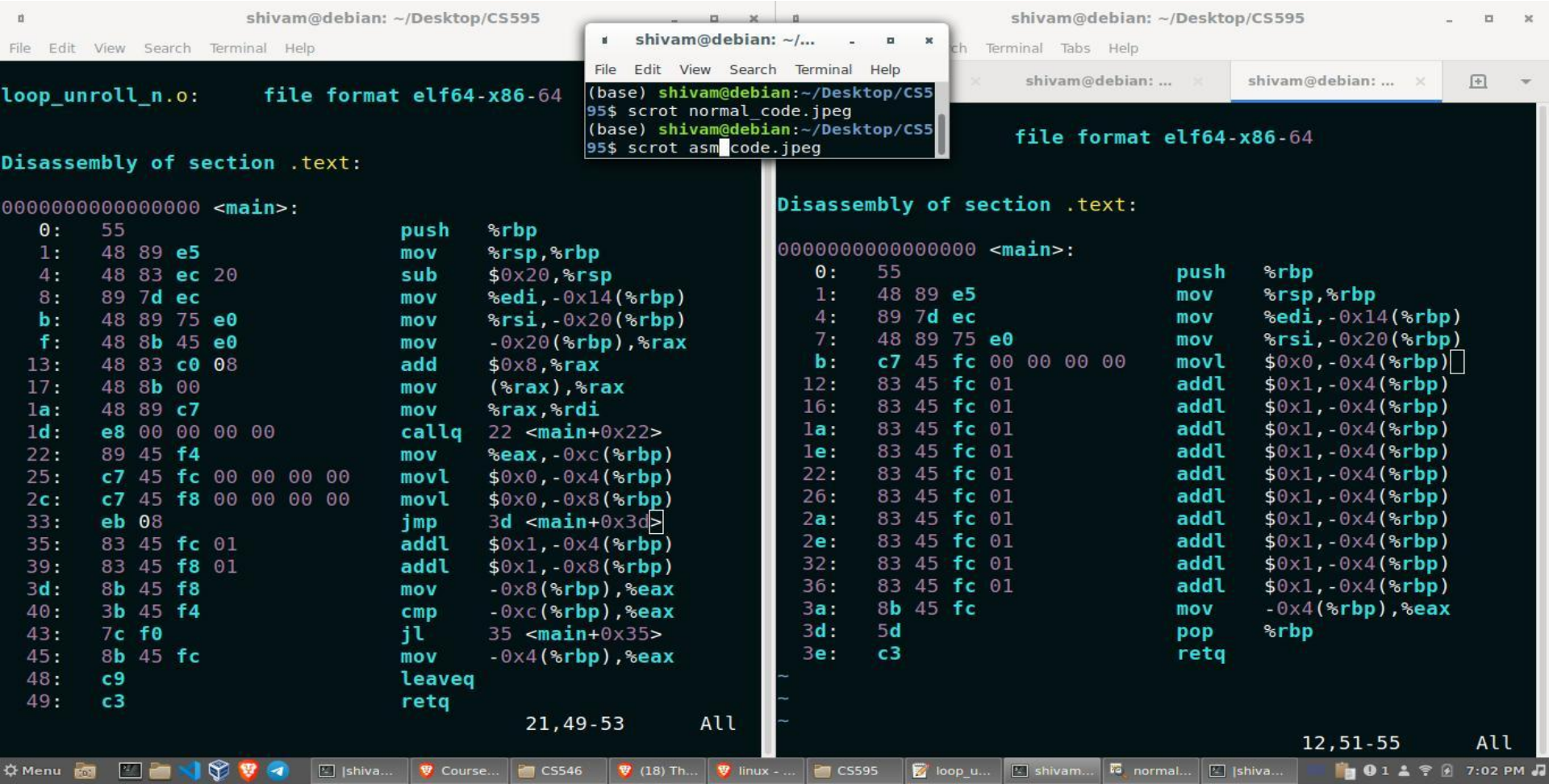
Optimization 1: removing conditionals, Can YOU write code without If statements or <, <=, >, >=, ?

- Using runtime information, create machine code that does things like loop unrolling.
- If loops are unrolled, then there doesn't exist the need to to increment a counter or check if its less than a bound.
- Typically, looping over data where the data has indeterminate type requires checking its type before doing an operation.
- Modern CPUs implement a feature called "Branch Prediction" which assume a conditional is true, hence they "conditional jump" to the segment of machine code that it would imply.
- Only a few clock cycles later would it realize it made a mistake and reverse, hence clogging up the throughput.

Loop unrolling example



Loop unrolling continued



Optimization 2: removing memory loads, when can you get around memory access penalty?

- Typically, when memory is written into a variable, and an instruction immediately follows that wants to read it, the processor has to stall until the writeback stage of the pipeline occurs. Similar for write to memory then read from memory.
- Based on runtime information, if we know the structure data, then we can add compiler optimizations to remove excessive memory operations, as well remove conditionals that do checks.
- Ex: suppose we know our data has 9 types, and our tables have at most 4, we can use the same function but remove the checks as well as excessive reads to repeated intermediate variables.

```
void MaterializeTuple(char* tuple) {
  for (int i = 0; i < num_slots_; ++i)
{
    char* slot = tuple + offsets_[i];
    switch(types_[i]) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;
      case FLOAT: …
      case STRING: …
      // etc.
    }
  }
}
```

interpreted

```
void MaterializeTuple(char* tuple) {
  *(tuple + 0) = ParseInt();    // i = 0
  *(tuple + 4) = ParseBoolean();// i = 1
  *(tuple + 5) = ParseInt();    // i = 2
}
```

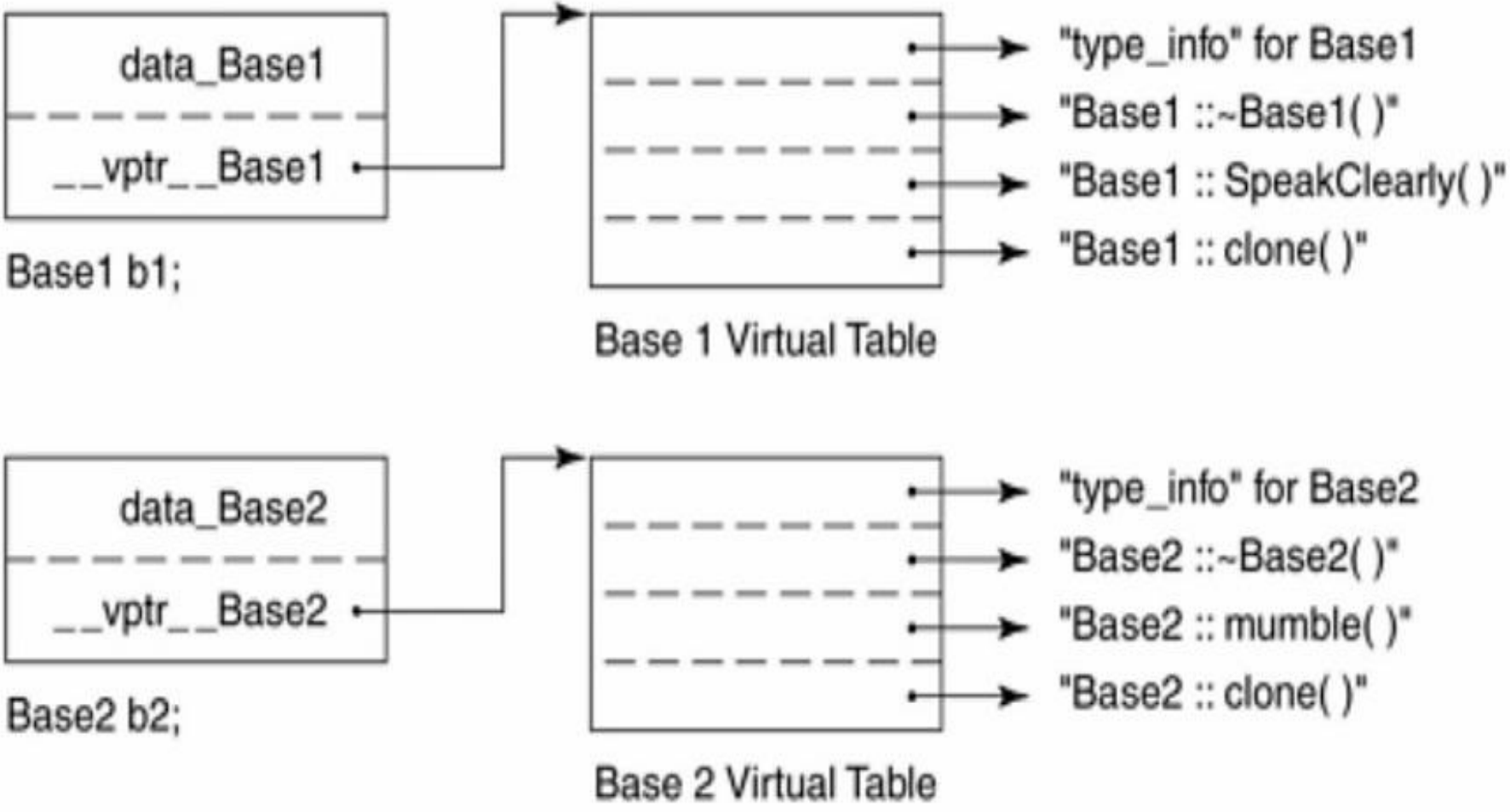
codegen’d

Basic five-stage pipeline

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

Optimization 3: Inline virtual functions

- In C++ virtual(abstract in java) is a keyword for class methods that allow for a child class to implement it and use at a later point in time.
- The issue is, suppose the method depends on a template(generic) that isn't determined until runtime, then without code generation the compiler will have to maintain a vtable(virtual function table) in the object or globally to index at runtime to jump to the "real" function address.
- So to prevent excessive instructions (indirection and conditional jumps), a JIT compiled version that has runtime information to determine "which version" can be used to inline a particular instantiation of a virtual function.


data_Base1
__vptr__Base1
Base1 b1;
"type_info" for Base1
"Base1 ::~Base1( )"
"Base1 :: SpeakClearly( )"
"Base1 :: clone( )"
Base 1 Virtual Table
data_Base2
__vptr__Base2
Base2 b2;
"type_info" for Base2
"Base2 ::~Base2( )"
"Base2 :: mumble( )"
"Base2 :: clone( )"
Base 2 Virtual Table

```
Base::num 100
Base::prv_data 100
&Base::num 0x7fff64d3ab78
&Base::prv_data 0x7fff64d3ab7c

Derived::num 200
Derived::prv_data 200
&Derived::num 0x7fff64d3ab90
&Derived::prv_data 0x7fff64d3ab94
Base::num 100
Base::prv_data 100
&Base::num 0x7fff64d3ab88
&Base::prv_data 0x7fff64d3ab8c

Pointer in Virutal Table: 0x400b8c
 Base::f()
Pointer in Virutal Table: 0x400bb6
 Base::g()
Pointer in Virutal Table: 0x400be0
 Base::h()

Pointer in Virutal Table: 0x400d66
 Derived::f()
Pointer in Virutal Table: 0x400d90
 Derived::g()
Pointer in Virutal Table: 0x400be0
 Base::h()
Pointer in Virutal Table: 0x400d12
 Derived::x()
Pointer in Virutal Table: 0x400d3c
 Derived::y()
```

Putting it all together (kind of)

- LLVM has whats called IR (intermediate representation) which is a pseudo assembly code with its own virtual instruction set.
- LLVM IR can be generated via either IRBUILDER api to do so programmatically(more difficult, error prone, potentially less optimized) or via compiler front ends that compile to IR for later use in JIT compilation.
- For C++ clang allows compilation of C++ code to LLVM IR(does gcc?)
- Impala does the second, i.e. compiling to LLVM via clang then injecting JIT compiled into process at runtime with runtime optimizations made that classically would only be possible at compile time.

Idea sounds nice in theory but...

- The paper claims their implementation has so far managed to replace interpreted (virtual) function calls with inlined specific ones and that they haven't implemented the other 2 optimizations.
- Arguably optimization #3 is the weakest of the 3, since in principle its preventing only a few extra instructions, compared to what would be saved by removing inner loops, conditional jumps, and memory read/write stalls.
- They claim they're working on optimization 1 and 2 and the ability to do this to pre compiled functions(they didnt explain how, perhaps functions from a shared library, so code can be borrowed and modified with runtime info and then dynamically linked? .so/.dll/.dylib depending on linux/windows/mac)

Support for user defined functions

- Impala provides support for users to provide UDFs
- rather than writing a function, compiling it to a shared object file, and then dynamically linking at runtime,
- They suggest it be compiled to IR via clang, and then JIT compiled with runtime info to allow UDFs to have same performance as native functions. Perhaps a detail they left out about the "precompiled" functions from the previous slide, is they be compiled to IR on the system when Impala is installed, and then are susceptible to the inlining and future optimizations.

Performance

	# Instructions	# Branches
Code generation disabled	72,898,837,871	14,452,783,201
Code generation enabled	19,372,467,372	3,318,983,319
Speedup	4.29x	3.76x

Table 4: Instruction and branch counts for TPCH-Q1

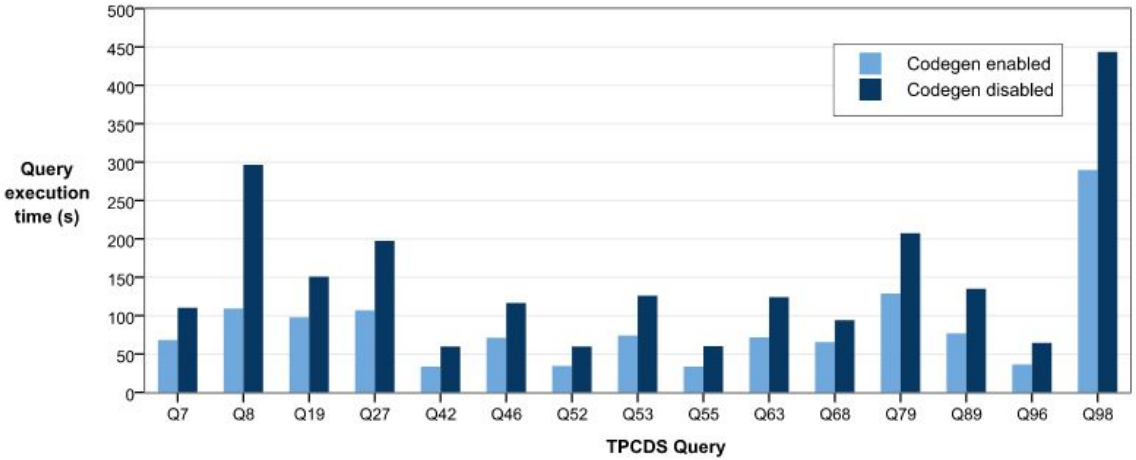


Figure 4: TPC-DS query execution with and without code generation

Interesting final thought.

- Clearly this paper is old and not fully implemented.
- In a fully implemented version of Impala with loops, conditional jumps, and pipeline stalling memory reads/writes optimized away, thing how its achieved.
- In principle its by adding runtime constants to individual instructions, and writing more instructions to get rid of loops and conditional jumps
- This results in a large number of instructions being added. Think in terms of opcodes, register ids and runtime constants. At minimum log2(#Opcodes) + 2 for the registers plus possibly upto 8 bytes each for a instruction constant at best? So code segment and data segment of process probably increase massively.

The End.
Questions?