# Runtime Code Generation in Cloudera Impala

Skye Wanderman-Milne
skye@cloudera.com

Nong Li
nong@cloudera.com

**Abstract**

*In this paper we discuss how runtime code generation can be used in SQL engines to achieve better query execution times. Code generation allows query-specific information known only at runtime, such as column types and expression operators, to be used in performance-critical functions as if they were available at compile time, yielding more efficient implementations. We present Cloudera Impala, an open-source, MPP database built for Hadoop, which uses code generation to achieve up to 5x speedups in query times.*

## 1 Introduction

Cloudera Impala is an open-source MPP database built for the Hadoop ecosystem. Hadoop has proven to be a very effective system to store and process large amounts of data using HDFS and HBase as the storage managers and MapReduce as the processing framework. Impala is designed to combine the flexibility and scalability that is expected from Hadoop with the performance and SQL support offered by commercial MPP databases. Impala currently executes queries 10-100x faster than existing Hadoop solutions and comparably to commercial MPP databases [1], allowing end users to run interactive, exploratory analytics on big data.

Impala is built from ground up to take maximal advantage of modern hardware and the latest techniques for efficient query execution. Impala is designed for analytic workloads, rather than OLTP, meaning it's common to run complex, long-running, CPU-bound queries. Runtime code generation using LLVM [3] is one of the techniques we use extensively to improve execution times. LLVM is a compiler library and collection of related tools. Unlike traditional compilers that are implemented as stand-alone applications, LLVM is designed to be modular and reusable. It allows applications like Impala to perform JIT compilation within a running process, with the full benefits of a modern optimizer and the ability to generate machine code for a number of architectures, by exposing separate APIs for all steps of the compilation process.

Impala uses LLVM to generate fully-optimized query-specific functions at runtime, which offer better performance than general-purpose precompiled functions. This technique can improve execution times by 5x or more for representative workloads. In this paper we describe how this achieved. Section 2 discusses how runtime code generation can be used to produce faster functions. Section 3 describes how we implement the code generation. Section 4 describes the implications of code generation for user-defined functions (UDFs). Section 5 details our results, and we conclude in Section 6.

*Copyright 2014 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.*
**Bulletin of the IEEE Computer Society Technical Committee on Data Engineering**

```
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

Figure 1: Example function illustrating runtime optimizations possible with code generation

## 2 Advantages of runtime code generation

Impala uses runtime code generation to produce query-specific versions of functions that are critical to performance. In particular, code generation is applied to "inner loop" functions, i.e., those that are executed many times in a given query, and thus constitute a large portion of the total time the query takes to execute. For example, a function used to parse a record in a data file into Impala's in-memory tuple format must be called for every record in every data file scanned. For queries scanning large tables, this could be trillions of records or more. This function must therefore be extremely efficient for good query performance, and even removing a few instructions from the function's execution can result in large query speedups.

Without code generation, inefficiencies in function execution are almost always necessary in order to handle runtime information not known at compile time. For example, a record-parsing function that only handles integer types will be faster at parsing an integer-only file than a function that handles other data types such as strings and floating-point numbers as well. However, the schemas of the files to be scanned are unknown at compile time, and so the most general-purpose function must be used, even if at runtime it is known that more limited functionality is sufficient.

Code generation improves execution by allowing for runtime variables to be used in performance-critical functions as if they were available at compile time. The generated function omits the overhead needed to interpret runtime-constant variables. Figure 1 gives an example of this. The figure illustrates the advantages of using code generation on an example record-parsing function MaterializeTuple() (as in the example given above, the function parses a record and materializes the result into an in-memory tuple). On the left-hand side of the figure is pseudocode for the interpreted function, i.e., the function that is implemented without code generation. On the right-hand side is pseudocode for a possible code-generated version of the same function. Note that the interpreted function is appropriate for any query, since it makes no assumptions about runtime information, whereas the code-generated function is specific to a certain resolution of runtime information. The code-generated MaterializeTuple() function for a different query could be different.

```
IntVal my_func(const IntVal& v1, const IntVal& v2) {
  return IntVal(v1.val * 7 / v2.val);
}

SELECT my_func(col1 + 10, col2) FROM ...
```

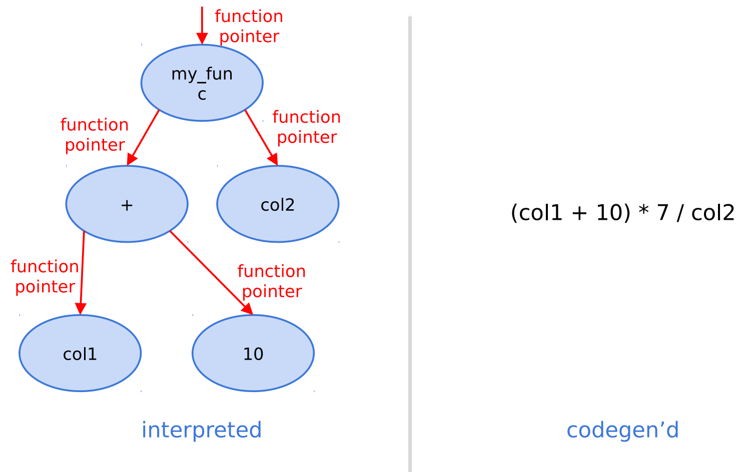


Figure 2: Expression tree optimization

More specifically, code generation allows us to optimize functions via the following techniques:

**Removing conditionals:** Runtime information that must be handled via an if or switch statement in an interpreted function can be resolved to a single case in the code-generated version, since the value of conditional is known at runtime. This is one of the most effective runtime optimizations, since branch instructions in the final machine code hinder instruction pipelining and instruction-level parallelism. By unrolling the for loop (since we know the number of iterations at runtime) and resolving the types, the branch instructions can be removed altogether.

**Removing loads:** Loading values from memory can be an expensive and pipeline-blocking operation. If the result of a load varies each time the function is invoked (e.g., loading the value of a slot in a tuple), there is nothing that can be done. However, if we know the load will always yield the same value on every invocation of the function, we can use code generation to substitute the load for the value. For example, in Figure 1, the offsets_ and types_ arrays are generated at the beginning of the query and do not vary, i.e., they are query-constant. Thus, in the code-generated version of the function, the values of these arrays can be directly inlined after unrolling the for loop.

**Inlining virtual function calls:** Virtual function calls incur a large performance penalty, especially when the called function is very simple, as the calls cannot be inlined. If the type of the object instance is known at runtime, we can use code generation to replace the virtual function call with a call directly to the correct function, which can then be inlined. This is especially valuable when evaluating expression trees. In Impala (as in many systems), expressions are composed of a tree of individual operators and functions, as illustrated in the left-hand side of Figure 2. Each type of expression that can appear in a tree is implemented by overriding a virtual function in the expression base class, which recursively calls its children expressions. Many of these expression functions are quite simple, e.g., adding two numbers. Thus, the cost of calling the virtual function often far exceeds the cost of actually evaluating the function. As illustrated in Figure 2, by resolving the virtual function calls with code generation and then inlining the resulting function calls, the expression tree can be evaluated directly with no function call overhead. In addition, inlining functions increases instruction-level parallelism, and allows the

compiler to make further optimizations such as subexpression elimination across expressions.

# 3 Generating code with LLVM

When a query plan is received by Impala's backend execution engine, LLVM is used to generate and compile query-specific versions of performance-critical functions before the query execution begins. This section explains in detail how the functions are generated before being compiled.

## 3.1 LLVM IR

LLVM primarily uses an intermediate representation (IR) for all parts of code generation. LLVM IR [8] resembles assembly language, being composed of a number of simple instructions that often have direct mappings to machine code. Frontends to LLVM, such as the Clang C++ compiler [5], generate IR, which can then be optimized and lowered to machine code by LLVM. We use two techniques for generating IR functions in Impala: using LLVM's IRBuilder, which allows for programmatically generating IR instructions, and cross-compiling C++ functions to IR using Clang.

## 3.2 IRBuilder

LLVM includes an IRBuilder class [7] as part of their C++ API. The IRBuilder is used to programmatically generate IR instructions, and as such can be used to assemble a function instruction by instruction. This is akin to writing functions directly in assembly: it's very simple, but can be quite tedious. In Impala, the C++ code for generating an IR function using the IRBuilder is generally many times longer than the C++ code implementing the interpreted version of the same function. However, this technique can be used to construct any function.

## 3.3 Compilation to IR

Instead of using the IRBuilder to construct query-specific functions, we generally prefer to compile a C++ function to IR using Clang, then inject query-specific information into the function at runtime. This allows us to write functions in C++ rather than constructing them instruction by instruction using the IRBuilder. We also cross-compile the functions to both IR and native code, allowing us to easily run either the interpreted or code-generated version. This is useful for debugging: we can isolate whether a bug is due to code generation or the function itself, and the native functions can be debugged using gdb.

Currently, our only mechanism for modifying compiled IR function is to replace function calls to interpreted functions with calls to equivalent query-specific generated functions. This is how we remove virtual functions calls as described in Section 2. For example, we cross compile many of the virtual functions implementing each expression type to both IR and native code. When running with code generation disabled, the native functions are run as-is, using the interpreted general-purpose expression implementations. When code generation is enabled, we recursively find all calls to child expressions and replace them with calls to code-generated functions.

Of course, this technique alone doesn't allow us to take full advantage of code generation. It doesn't help us with many of the techniques described in section 2, such as removing conditionals and loads. For now we generate functions that benefit from these techniques using the IRBuilder. However, we are currently developing a new framework for modifying precompiled IR functions. Returning to to the MaterializeTuple() example in Figure 1, the main optimizations we would like to perform on the interpreted code in order to take advantage of runtime information are (1) to unroll the for loop using the known num_slots_ variable, so we can replace each iteration with iteration-specific runtime information, and (2) to replace accesses of offsets_ and types_ with the actual values. Once we have a framework for these transformations, we will be able to implement code-generated functions more easily and quickly than we can with the IRBuilder.

| Query | Code generation disabled | Code generation enabled | Speedup |
|---|---|---|---|
| select count(*) from lineitem | 3.554 sec | 2.976 sec | 1.19x |
| select count(l_orderkey) from lineitem | 6.582 sec | 3.522 sec | 1.87x |
| TPCH-Q1 | 37.852 sec | 6.644 sec | 5.70x |

Table 3: Query times with and without code generation

## 4 User-defined functions

Impala provides a C++ user-defined function (UDF) API, and the most conventional method for authoring a UDF is to implement it in C++ and compile it to a shared object, which is dynamically linked at runtime. However, as discussed above, Impala can compile and execute IR functions generated by Clang. We take advantage of this functionality to execute UDFs compiled to IR, rather than to a shared object. This enables inlining function calls across user functions, meaning UDFs can have identical performance to Impala's built-ins.

In addition to performance benefits, this architecture easily allows UDFs to be authored in other languages. Just as we use Clang to compile C++ functions to IR, any language with an LLVM frontend can be used to author UDFs without modification to the query engine. For example, Numba [9] allows compilation from Python to IR. Using our approach, a developer could author UDFs in Python that would be even more performant than C++ UDFs statically compiled to shared objects.

## 5 Experimental results

In Table 3, we show the effectiveness of runtime code generation as we increase the complexity of the query. These queries were run on a 10-node cluster over a 600M-row Avro [4] data set. In the first query, we do a simple count of the rows in the table. This doesn't require actually parsing any Avro data, so the only benefit of code generation is to improve the efficiency of the count aggregation, which is already quite simple. The resulting speedup from code generation is thus quite small. In the second query, we do a count of a single column, which requires parsing the Avro data in order to detect null values. This increases the benefit of code generation by 60% over the simpler query. Finally, we run the TPCH-Q1 query, which is reproduced in Figure 3. This involves multiple aggregates, expressions, and group by clauses, resulting in a much larger speedup.

```
select
  l_returnflag, l_linestatus, sum(l_quantity), sum(l_extendedprice),
  sum(l_extendedprice * (1 - l_discount)),
  sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
  avg(l_quantity), avg(l_extendedprice), avg(l_discount), count(1)
from lineitem
where l_shipdate<='1998-09-02'
group by l_returnflag, l_linestatus
```

Figure 3: TPCH-Q1 query

In Table 4, we look at how runtime code generation reduces the number of instructions executed when running TPCH-Q1. These counts were collected from the hardware counters using the Linux perf tool. The counters are collected for the entire duration of the query and include code paths that do not benefit from code generation.

|  | # 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

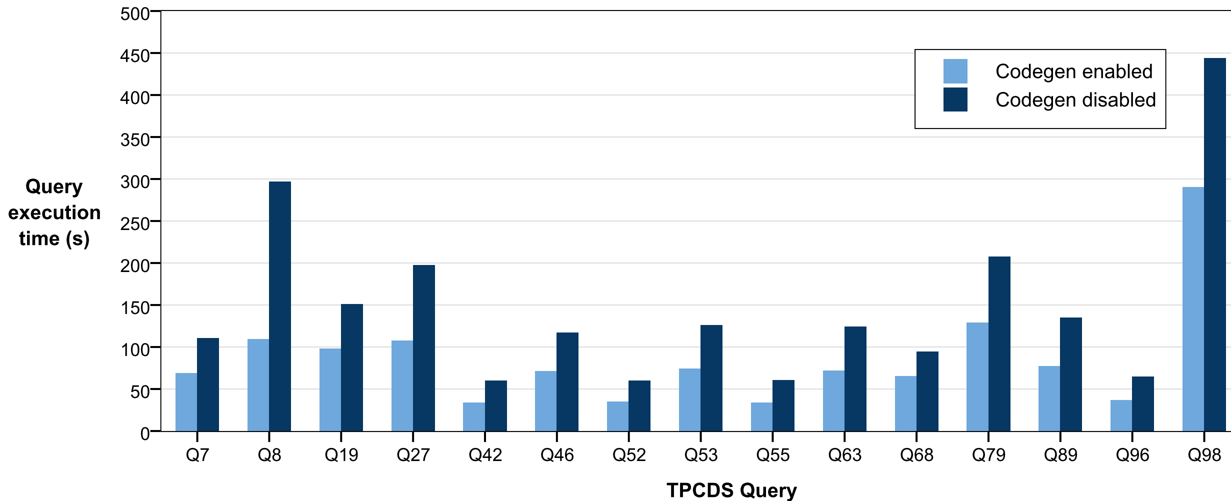


Figure 4: TPC-DS query execution with and without code generation

Finally, in Figure 4, we present the results of running several slightly-modified TPC-DS queries with and without code generation. These queries were run on a 10-node cluster over a 1-terabyte scale factor Parquet [6, 2] dataset. These queries do not achieve the 5x speedup seen on the above TPCH-Q1 result due to some sections of the query execution not implementing code generation. In particular, nearly all the TPC-DS queries contain an ORDER BY clause, which does not currently benefit from code generation, and the Parquet file parser does not currently use code generation (we used Parquet rather than Avro, which does use code generation, for these results because otherwise the queries are IO-bound). However, there is still a significant speedup gained on every query, and we expect this to become much larger once more of the execution can be code generated.

## 6 Conclusions and future work

LLVM allows us to implement a general-purpose SQL engine where individual queries perform as if we had written a dedicated application specifically for that query. Code generation has been available in Impala since the initial release, and we're excited about the ideas we have to further improve it. We currently only code generate functions for certain operators and functions, and are expanding our efforts to more of the execution. The more of the query that is generated, the more we can take advantage of function inlining and subsequent inter-function optimizations. Eventually we would like the entire query execution tree to be collapsed into a single function, in order to eliminate almost all in-memory accesses and keep state in registers.

We are also working on a project integrating a Python development environment with Impala, taking advantage of Impala's ability to run IR UDFs. The project will allow users, working completely within the Python shell or scripts, to author UDFs and run them across a Hadoop cluster, with parameters and results being auto-

matically converted between native Python and Impala types as needed. We hope this will enable new uses cases involving scientific computing for Impala and Hadoop.

These are just a few of the many possible directions we can take with runtime code generation. The technique has proven enormously useful, and we imagine it will become more widespread in performance-critical applications.

## References


[1] Erickson, Justin, Greg Rahn, Marcel Kornacker, and Yanpei Chen. *"Impala Performance Update: Now Reaching DBMS-Class Speed."* Cloudera Developer Blog. Cloudera, 13 Jan. 2014. `http://blog.cloudera.com/blog/2014/01/impala-performance-dbms-class-speed/`

[2] *"Introducing Parquet: Efficient Columnar Storage for Apache Hadoop."* Cloudera Developer Blog. Cloudera, 13 March 2013. `http://blog.cloudera.com/blog/2013/03/introducing-parquet--columnar-storage-for-apache-hadoop/`

[3] "LLVM: An Infrastructure for Multi-Stage Optimization", Chris Lattner. *Masters Thesis*, Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[4] `http://avro.apache.org/`

[5] `http://clang.llvm.org/`

[6] `http://parquet.io/`

[7] `http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html`

[8] `http://llvm.org/docs/LangRef.html`

[9] `http://numba.pydata.org`