Building a Modern Database Using LLVM

Skye Wanderman-Milne, Cloudera
skye@cloudera.com

LLVM Developers' Meeting, Nov. 6-7

Overview

- What is Cloudera Impala?
- Why code generation?
- Writing IR vs. cross compilation
- Results

What is Cloudera Impala?

- High-performance distributed SQL engine for Hadoop
  - Similar to Google's Dremel
  - Designed for analytic workloads
- Reads/writes data from HDFS, HBase
  - Schema on read
  - Queries data directly from supported formats: text (CSV), Avro, Parquet, and more
- Open-source (Apache licensed)

What is Cloudera Impala?

- Primary goal: SPEED!
- Uses LLVM to JIT compile query-specific functions

Why code generation?

Code generation (codegen) lets us use query-specific information to do less work

- Remove conditionals
- Propagate constant offsets, pointers, etc.
- Inline virtual functions calls

```
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

```
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

```
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

```
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
}
```

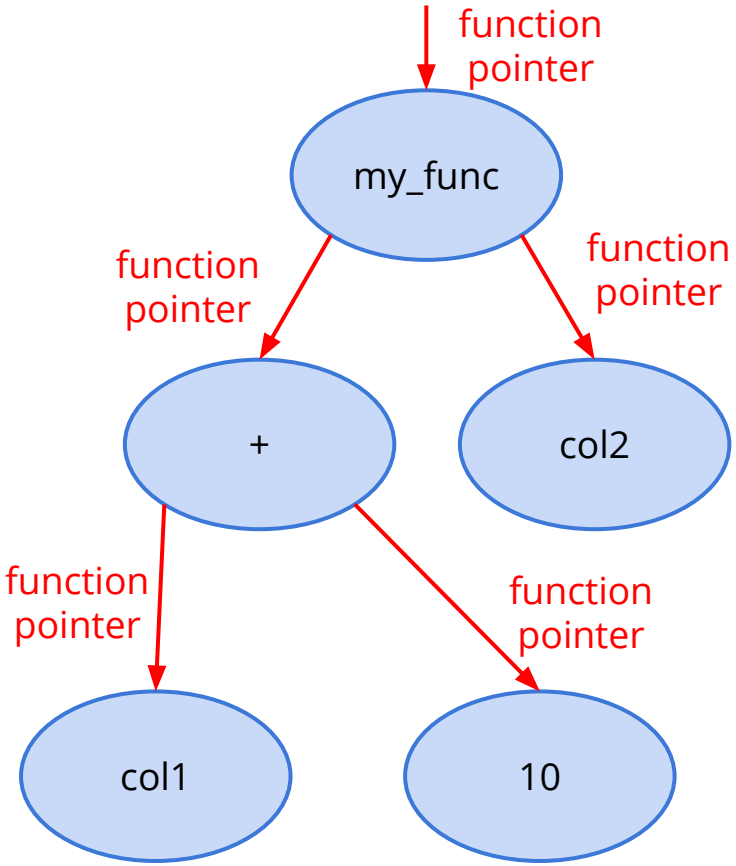
codegen’d

User-Defined Functions (UDFs)

- Allows users to extend Impala's functionality by writing their own functions
  e.g. `select my_func(c1) from table;`
- Defined as C++ functions
- UDFs can be compiled to IR (vs. native code) with Clang $\Rightarrow$ inline UDFs

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

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



interpreted

(col1 + 10) * 7 / col2

codegen’d

User-Defined Functions (UDFs)

Future work: UDFs in other languages with LLVM frontends

Two choices for code generation:

- Use the C++ API to handcraft IR
- Compile C++ to IR

```
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

```
void HdfsAvroScanner::MaterializeTuple(MemPool* pool, uint8_t** data, Tuple* tuple) {
  BOOST_FOREACH(const SchemaElement& element, avro_header_->schema) {
    const SlotDescriptor* slot_desc = element.slot_desc;
    bool write_slot = false;
    void* slot = NULL;
    PrimitiveType slot_type = INVALID_TYPE;
    if (slot_desc != NULL) {
      write_slot = true;
      slot = tuple->GetSlot(slot_desc->tuple_offset());
      slot_type = slot_desc->type();
    }

    avro_type_t type = element.type;
    if (element.null_union_position != -1
        && !ReadUnionType(element.null_union_position, data)) {
      type = AVRO_NULL;
    }

    switch (type) {
      case AVRO_NULL:
        if (slot_desc != NULL) tuple->SetNull(slot_desc->null_indicator_offset());
        break;
      case AVRO_BOOLEAN:
        ReadAvroBoolean(slot_type, data, write_slot, slot, pool);
        break;
      case AVRO_INT32:
        ReadAvroInt32(slot_type, data, write_slot, slot, pool);
        break;
      case AVRO_INT64:
        ReadAvroInt64(slot_type, data, write_slot, slot, pool);
        break;
      case AVRO_FLOAT:
        ReadAvroFloat(slot_type, data, write_slot, slot, pool);
        break;
      case AVRO_DOUBLE:
        ReadAvroDouble(slot_type, data, write_slot, slot, pool);
        break;
      case AVRO_STRING:
      case AVRO_BYTES:
        ReadAvroString(slot_type, data, write_slot, slot, pool);
        break;
      default:
        DCHECK(false) << "Unsupported SchemaElement: " << type;
    }
  }
}
```

Native interpreted function

```
Function* HdfsAvroScanner::CodegenMaterializeTuple(HdfsScanNode* node,
                                                   LlvmCodeGen* codegen) {
  const string& table_schema_str = node->hdfs_table()->avro_schema();

  // HdfsAvroScanner::Codegen() (which calls this function) gets called by
HdfsScanNode
  // regardless of whether the table we're scanning contains Avro files or not. If
this
  // isn't an Avro table, there is no table schema to codegen a function from (and
there's
  // no need to anyway).
  // TODO: HdfsScanNode shouldn't codegen functions it doesn't need.
  if (table_schema_str.empty()) return NULL;

  ScopedAvroSchemaT table_schema;
  int error = avro_schema_from_json_length(
      table_schema_str.c_str(), table_schema_str.size(), &table_schema.schema);
  if (error != 0) {
    LOG(WARNING) << "Failed to parse table schema: " << avro_strerror();      true);
    return NULL;
  }
  int num_fields = avro_schema_record_size(table_schema.schema);
  DCHECK_GT(num_fields, 0);

  LLVMContext& context = codegen->context();
  LlvmCodeGen::LlvmBuilder builder(context);

  Type* this_type = codegen->GetType(HdfsAvroScanner::LLVM_CLASS_NAME);
  DCHECK(this_type != NULL);
  PointerType* this_ptr_type = PointerType::get(this_type, 0);

  TupleDescriptor* tuple_desc = const_cast<TupleDescriptor*>(node->tuple_desc());
  StructType* tuple_type = tuple_desc->GenerateLlvmStruct(codegen);
  Type* tuple_ptr_type = PointerType::get(tuple_type, 0);

  Type* tuple_opaque_type = codegen->GetType(Tuple::LLVM_CLASS_NAME);
  PointerType* tuple_opaque_ptr_type = PointerType::get(tuple_opaque_type, 0);

  Type* data_ptr_type = PointerType::get(codegen->ptr_type(), 0); // char**
  Type* mempool_type = PointerType::get(codegen->GetType(MemPool::LLVM_CLASS_NAME),
0);

  LlvmCodeGen::FnPrototype prototype(codegen, "MaterializeTuple", codegen-
>void_type());
  prototype.AddArgument(LlvmCodeGen::NamedVariable("this", this_ptr_type));
  prototype.AddArgument(LlvmCodeGen::NamedVariable("pool", mempool_type));
  prototype.AddArgument(LlvmCodeGen::NamedVariable("data", data_ptr_type));
  prototype.AddArgument(LlvmCodeGen::NamedVariable("tuple", tuple_opaque_ptr_type));
  Value* args[4];
  Function* fn = prototype.GeneratePrototype(&builder, args);

  Value* this_val = args[0];
  Value* pool_val = args[1];
  Value* data_val = args[2];
  Value* opaque_tuple_val = args[3];

  Value* tuple_val = builder.CreateBitCast(opaque_tuple_val, tuple_ptr_type,
"tuple_ptr");

  // Codegen logic for parsing each field and, if necessary, populating a slot with
the
  // result.
  for (int field_idx = 0; field_idx < num_fields; ++field_idx) {
    avro_datum_t field =
        avro_schema_record_field_get_by_index(table_schema.schema, field_idx);
    SchemaElement element = ConvertSchemaNode(field);
    int col_idx = field_idx + node->num_partition_keys();
    int slot_idx = node->GetMaterializedSlotIdx(col_idx);

    // The previous iteration may have left the insert point somewhere else
    builder.SetInsertPoint(&fn->back());

    BasicBlock* endif_block = BasicBlock::Create(context, "endif", fn);
    Function* read_union_fn =
        codegen->GetFunction(IRFunction::READ_UNION_TYPE);
    Value* null_union_pos_val =
        codegen->GetIntConstant(TYPE_INT, element.null_union_position);
    Value* is_not_null_val = builder.CreateCall3(
        read_union_fn, this_val, null_union_pos_val, data_val, "is_not_null");
    builder.CreateCondBr(is_not_null_val, read_field_block, null_block);

    // Write branch at end of read_field_block, we fill in the rest later
    builder.SetInsertPoint(read_field_block);
    builder.CreateBr(endif_block);

    // Write null field IR
    builder.SetInsertPoint(null_block);
    if (slot_idx != HdfsScanNode::SKIP_COLUMN) {
      SlotDescriptor* slot_desc = node->materialized_slots()[slot_idx];
      Function* set_null_fn = slot_desc->CodegenUpdateNull(codegen, tuple_type,

      DCHECK(set_null_fn != NULL);
      builder.CreateCall(set_null_fn, tuple_val);
    }
    // LLVM requires all basic blocks to end with a terminating instruction
    builder.CreateBr(endif_block);
  } else {
    // Field is never null, read field unconditionally.
    builder.CreateBr(read_field_block);
  }

  // Write read_field_block IR starting at the beginning of the block
  builder.SetInsertPoint(read_field_block, read_field_block->begin());
  Function* read_field_fn;
  switch (element.type) {
    case AVRO_BOOLEAN:
      read_field_fn = codegen->GetFunction(IRFunction::READ_AVRO_BOOLEAN);
      break;
    case AVRO_INT32:
      read_field_fn = codegen->GetFunction(IRFunction::READ_AVRO_INT32);
      break;
    case AVRO_INT64:
      read_field_fn = codegen->GetFunction(IRFunction::READ_AVRO_INT64);
      break;
    case AVRO_FLOAT:
      read_field_fn = codegen->GetFunction(IRFunction::READ_AVRO_FLOAT);
      break;
    case AVRO_DOUBLE:
      read_field_fn = codegen->GetFunction(IRFunction::READ_AVRO_DOUBLE);
      break;
    case AVRO_STRING:
    case AVRO_BYTES:
      read_field_fn = codegen->GetFunction(IRFunction::READ_AVRO_STRING);
      break;
    default:
      DCHECK(false) << "Unsupported SchemaElement: " << element.type;
  }

  if (slot_idx != HdfsScanNode::SKIP_COLUMN) {
    // Field corresponds to materialized column
    SlotDescriptor* slot_desc = node->materialized_slots()[slot_idx];
    Value* slot_type_val = codegen->GetIntConstant(TYPE_INT, slot_desc->type());
    Value* slot_val =
        builder.CreateStructGEP(tuple_val, slot_desc->field_idx(), "slot");
    Value* opaque_slot_val =
        builder.CreateBitCast(slot_val, codegen->ptr_type(), "opaque_slot");
    Value* read_field_args[] =
        {this_val, slot_type_val, data_val, codegen->true_value(),
         opaque_slot_val, pool_val};
    builder.CreateCall(read_field_fn, read_field_args);
  } else {
    // Field corresponds to an unmaterialized column
    Value* dummy_slot_type_val = codegen->GetIntConstant(TYPE_INT, 0);
    Value* read_field_args[] =
      {this_val, dummy_slot_type_val, data_val, codegen-
>false_value(),
       dummy_slot_val, pool_val};
    builder.CreateCall(read_field_fn, read_field_args);
  }
  builder.SetInsertPoint(&fn->back());
  builder.CreateRetVoid();
  return codegen->FinalizeFunction(fn);
}
```

Codegen function

(uses C++ API to produce IR)

Cross-compilation

- Compile C++ functions to both native code and IR
- Native code useful for debugging
  - LLVM experts: can I debug JIT'd functions?
- Inject run-time information and inline IR
  - Convert interpreted function to codegen'd function

```
int HdfsAvroScanner::DecodeAvroData(int max_tuples, MemPool* pool, uint8_t** data,
                                    Tuple* tuple, TupleRow* tuple_row) {
  int num_to_commit = 0;
  for (int i = 0; i < max_tuples; ++i) {
    InitTuple(template_tuple_, tuple);
    MaterializeTuple(pool, data, tuple);
    tuple_row->SetTuple(scan_node_->tuple_idx(), tuple);
    if (ExecNode::EvalConjuncts(&(*conjuncts_)[0], num_conjuncts_, tuple_row)) {
      ++num_to_commit;
      tuple_row = next_row(tuple_row);
      tuple = next_tuple(tuple);
    }
  }
  return num_to_commit;
}
```

- Cross-compile to native code and IR
- When running native code, leave as is
- When using codegen, replace highlighted functions with query-aware equivalents

```
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

(native code)

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

codegen’d

(injected into IR)

```
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
(native code)

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

codegen’d
(injected into IR)

Can’t cross-compile native code to IR (efficiently):

- Loop may not be unrolled
- Can’t inline runtime information

```
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.
    }
  }
}
```

Native code

(can't be efficiently compiled to IR)

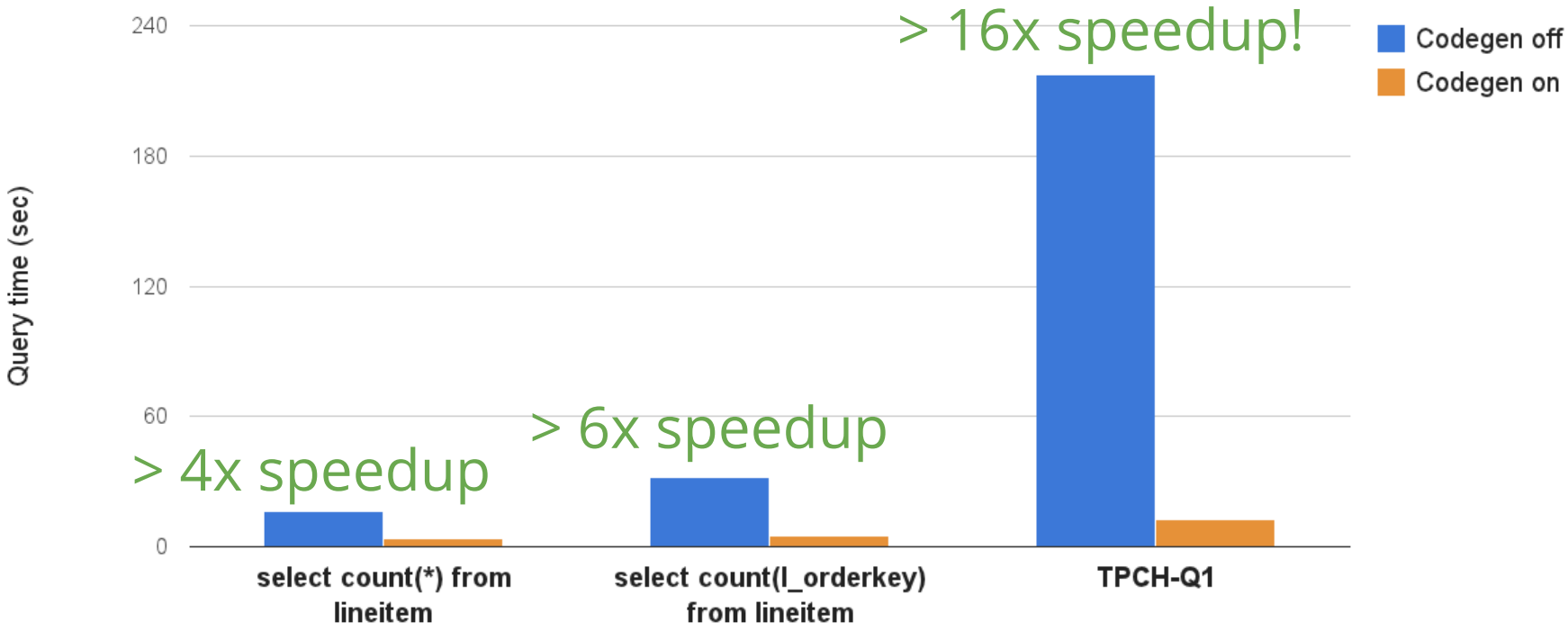
```
void MaterializeTuple(char* tuple) {
  XCOMPILE_FOR (int i, num_slots_) {
    char* slot = tuple +
        XCOMPILE_LOAD(offsets_, i);
    switch(XCOMPILE_LOAD(types_, i)) {
      case BOOLEAN:
        *slot = ParseBoolean();
        break;
      case INT:
        *slot = ParseInt();
        break;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

Theoretical cross-compiled code

(not yet implemented)

Results

Results



10 node cluster (12 disks / 48GB RAM / 8 cores per node)
~40 GB / ~60M row Avro dataset

TPCH-Q1

```
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
```

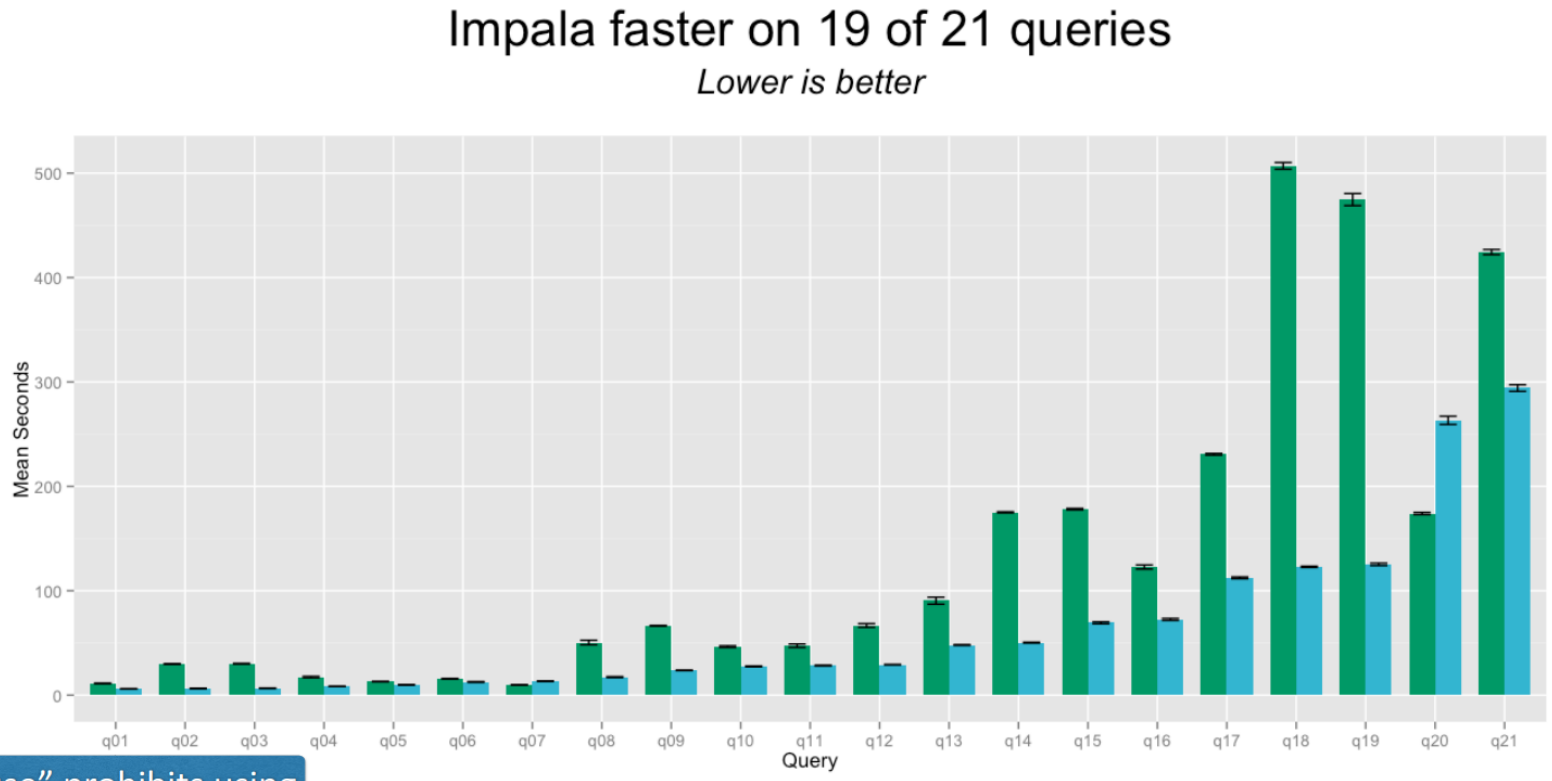
Results

	Query time	# Instructions	# Branches
Codegen off	7.55 sec	72,898,837,871	14,452,783,201
Codegen on	1.76 sec	19,372,467,372	3,318,983,319
Speedup	4.29x	3.76x	4.35x

Codegen takes ~150ms

TPCH-Q1
2.7GB / ~20M rows Avro dataset
Single desktop node

Results



"DeWitt Clause" prohibits using DBMS vendor name

Thank you!

https://github.com/cloudera/impala

skye@cloudera.com