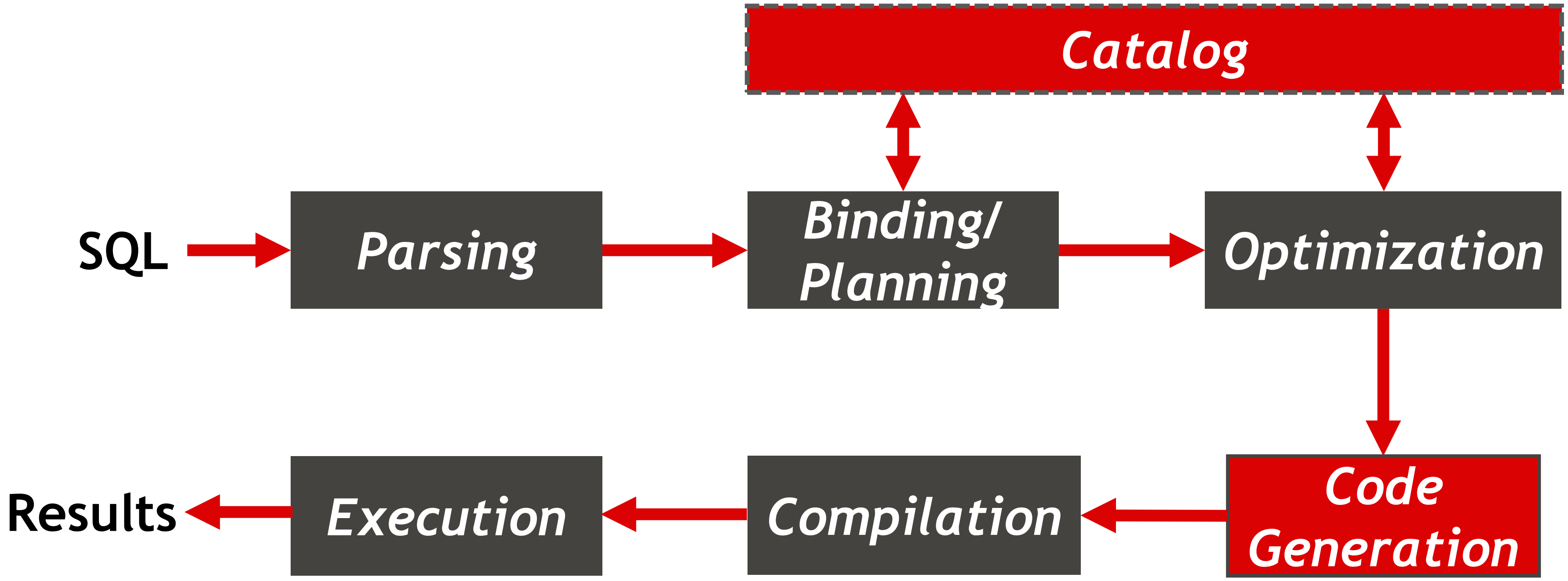
CODE GENERATION IN PELOTON

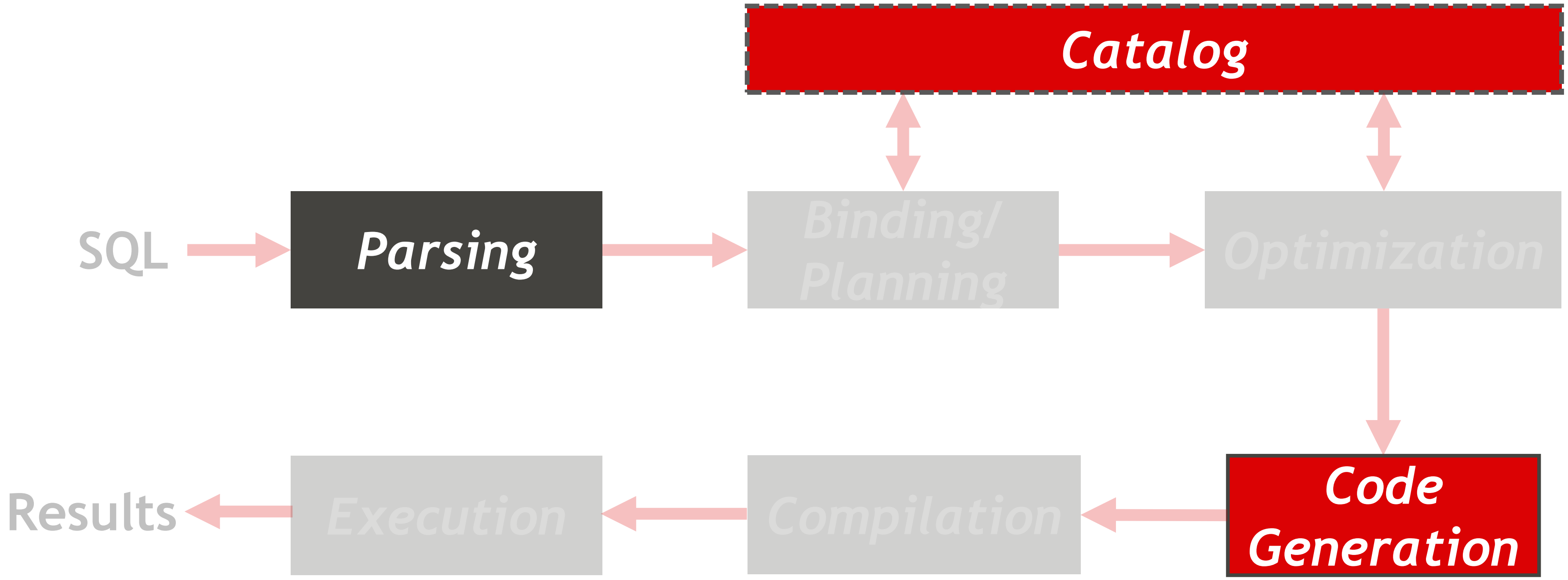
Prashanth Menon

23/01/2018

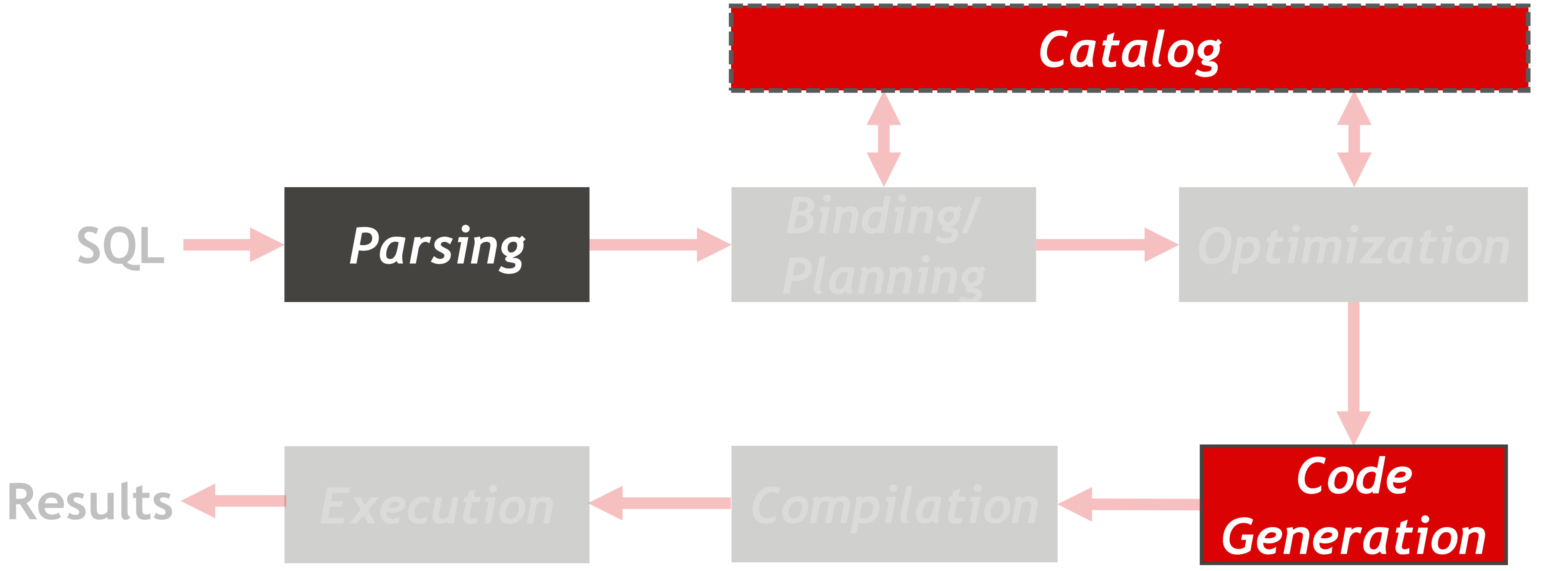
QUERY EXECUTION FLOW



QUERY EXECUTION FLOW

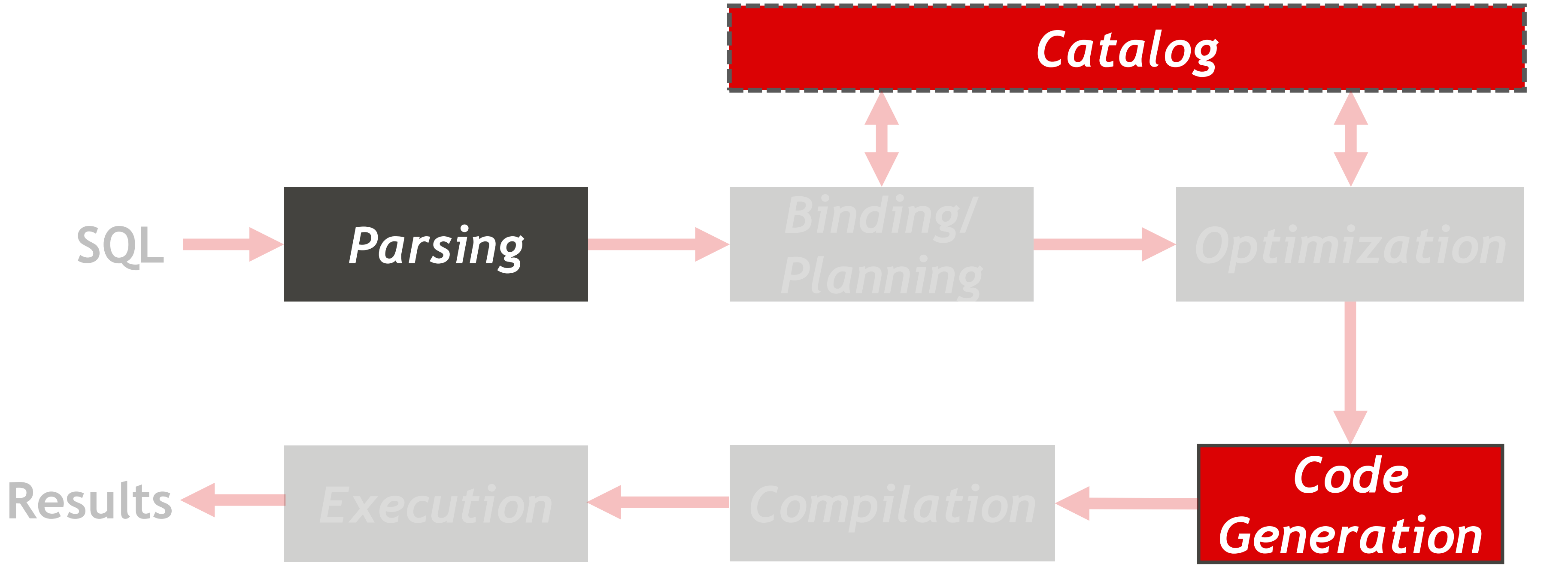


QUERY EXECUTION FLOW



F: PlanTree → IR

QUERY EXECUTION FLOW



F: IR→ x86

F: PlanTree → IR

CATALOG

- All built-in functions are registered statically in catalog
  - *More specifically, in '**catalog.cpp**'*

1. Built-ins are associated with an ***OperatorId***
   - *Add new entries to OperatorId enum for your builtins*
2. Add new catalog entry to **pg_proc** catalog
   - *Associate with newly added OperatorId*

CODE GENERATION

- All functions are associated with a **TypeSystem**

- A TypeSystem defines the interface for a SQL type
  - *Essentially a function pointer table*

1. Implement appropriate interface
   - *Unary, Binary, or N-ary function*
2. Create instance of function for your SQL type
3. Install instance in function pointer table

UNARY OPERATORS

- UPPER/LOWER are *unary* functions
  - *UnaryOperatorHandleNull takes care of NULL handling*

```
Value null_val, ret_val;
lang::If is_null{codegen, val.IsNull(codegen), "is_null"};
{
  // If the value is NULL, return the NULL value for the result type
  null_val = ResultType(val.GetType()).GetSqlType().GetNullValue(codegen);
}
is_null.ElseBlock();
{
  // If the input isn't NULL, perform the non-null-aware operation
  ret_val = Impl(codegen, val, ctx);
}
is_null.EndIf();
```

N-ARY OPERATOR

- CONCAT is an N-ary operator
  - *It can accept NULL inputs*
  - *You need to handle this*

- Use provided Concat implementation!

PROXIES

- You may want to call C/C++ to do work
  - *Do not perform regex compilation in codegen*
- Use macros to outline all (static) functions you want to call from codegen
- Use *CodeGen::Call(F, args...)* to invoke C/C++
- Only use C/C++ native types!
  - *Integer types, float, doubles, and pointers*
  - *Pointers to complex objects are okay, provided you have a proxy definition for it*

PROXIES EXAMPLE - ASCII

- Define your plain old C/C++ function:

```
uint32_t StringFunctions::Ascii(UNUSED_ATTRIBUTE executor::ExecutorContext &ctx,
                                const char *str, uint32_t length) {
  PL_ASSERT(str != nullptr);
  return length <= 1 ? 0 : static_cast<uint32_t>(str[0]);
}
```

PROXIES EXAMPLE - ASCII

- Declare a proxy

```
PROXY(StringFunctions) {
  // Proxy everything in function::StringFunctions
  DECLARE_METHOD(Ascii);
};
```

- Define the proxy:

```
DEFINE_METHOD(peloton::function, StringFunctions, Ascii);
```

PROXIES EXAMPLE - ASCII

- Usage:

```
struct Ascii : public TypeSystem::UnaryOperatorHandleNull {
  bool SupportsType(const Type &type) const override {
    return type.GetSqlType() == Varchar::Instance();
  }

  Type ResultType(UNUSED_ATTRIBUTE const Type &val_type) const override {
    return Integer::Instance();
  }

  Value Impl(CodeGen &codegen, const Value &val,
             const TypeSystem::InvocationContext &ctx) const override {
    llvm::Value *executor_ctx = ctx.executor_context;
    llvm::Value *raw_ret =
        codegen.Call(StringFunctionsProxy::Ascii,
                     {executor_ctx, val.GetValue(), val.GetLength()});
    return Value{Integer::Instance(), raw_ret};
  }
};
```

PROXIES EXAMPLE - ASCII

- Usage:

```
struct Ascii : public TypeSystem::UnaryOperatorHandleNull {
  bool SupportsType(const Type &type) const override {
    return type.GetSqlType() == Varchar::Instance();
  }

  Type ResultType(UNUSED_ATTRIBUTE const Type &val_type) const override {
    return Integer::Instance();
  }

  Value Impl(CodeGen &codegen, const Value &val,
             const TypeSystem::InvocationContext &ctx) const override {
    llvm::Value *executor_ctx = ctx.executor_context;
    llvm::Value *raw_ret =
        codegen.Call(StringFunctionsProxy::Ascii,
                     {executor_ctx, val.GetValue(), val.GetLength()});
    return Value{Integer::Instance(), raw_ret};
  }
};
```

Value

- All operators return a codegen::Value
  - *Values have a type, value, length and NULL bit*
- Almost the same API as type::Value
- Cannot mix codegen::Value and type::Value
  - *codegen::Value is a symbolic, compile-time representation of a SQL value*

TESTING

- Use psql to test through command line
- Write test case for your function
    - *Modify existing function unit tests*
- Use test scripts in ”testing/dml”
- Use LOGGING statements