Pragmatic Code Generation for Efficient Execution

Satyam Shekhar, NetSpring Inc

Agenda

- Introduction
- SQL Execution Primer
- Why generate code (or vectorize)?
- Code generation in NetSpring
- Practical considerations for generating code

Agenda

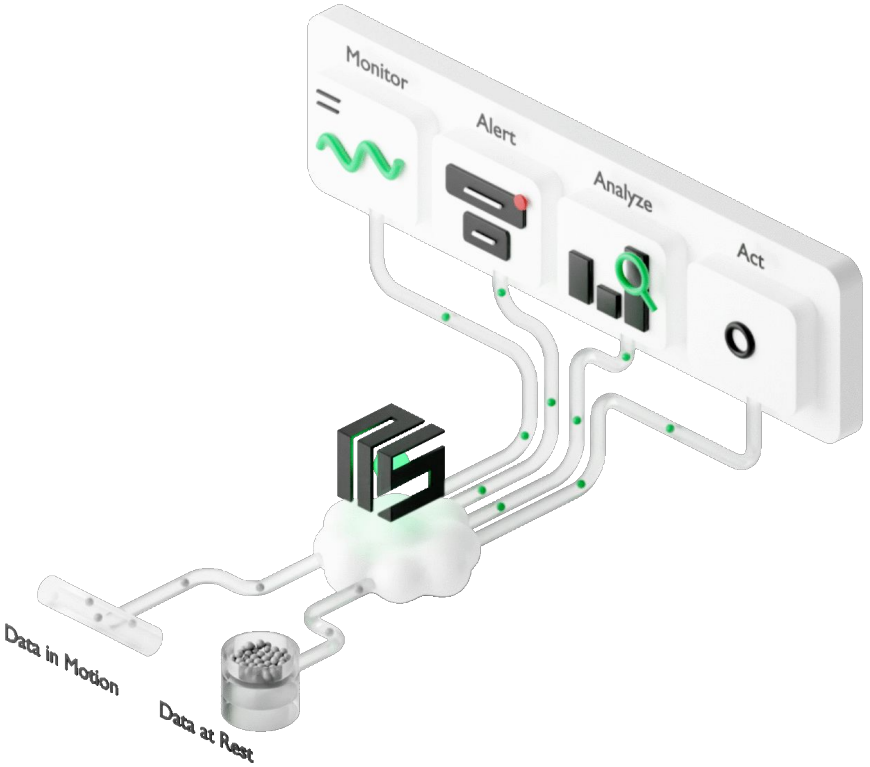
- **Introduction**
- SQL Execution Primer
- Why generate code (or vectorize)?
- Code generation in NetSpring
- Practical considerations for generating code

NetSpring

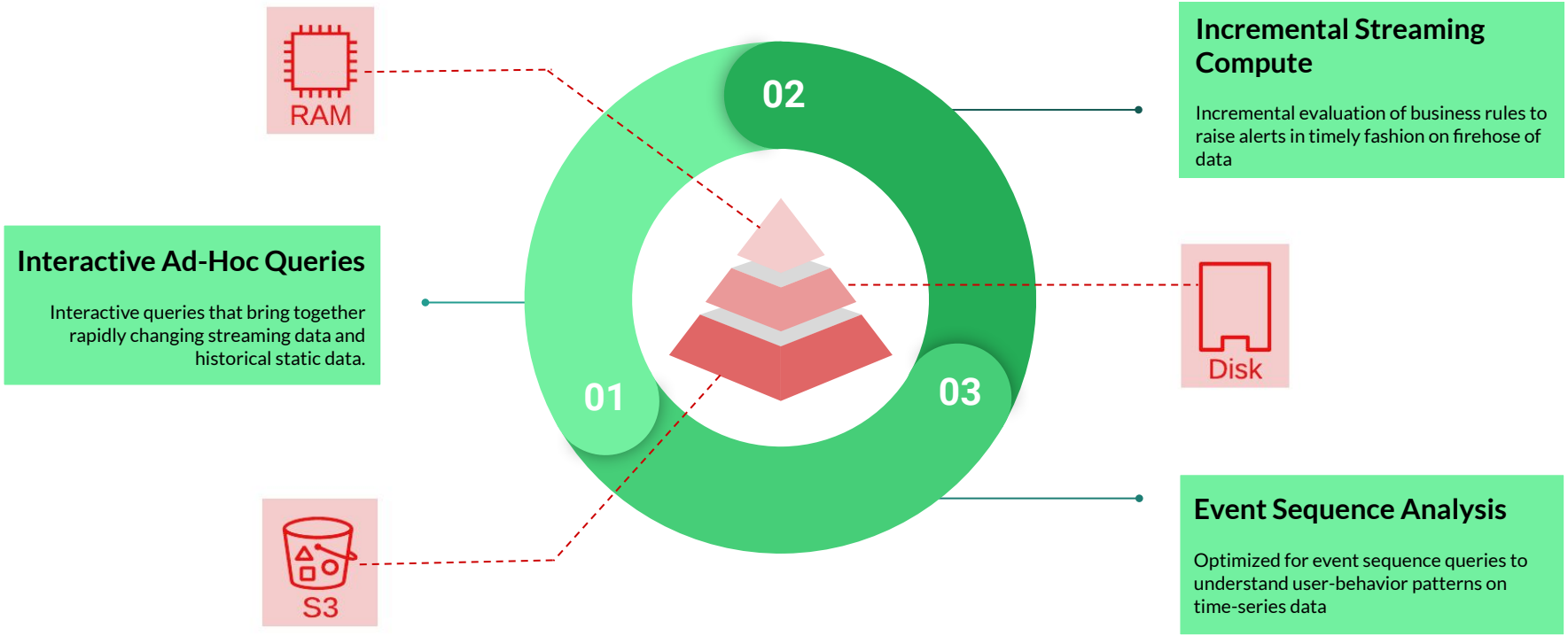
Continuous analytics on operational event data

Analyze across data in motion and at rest

Cloud platform-as-a-service



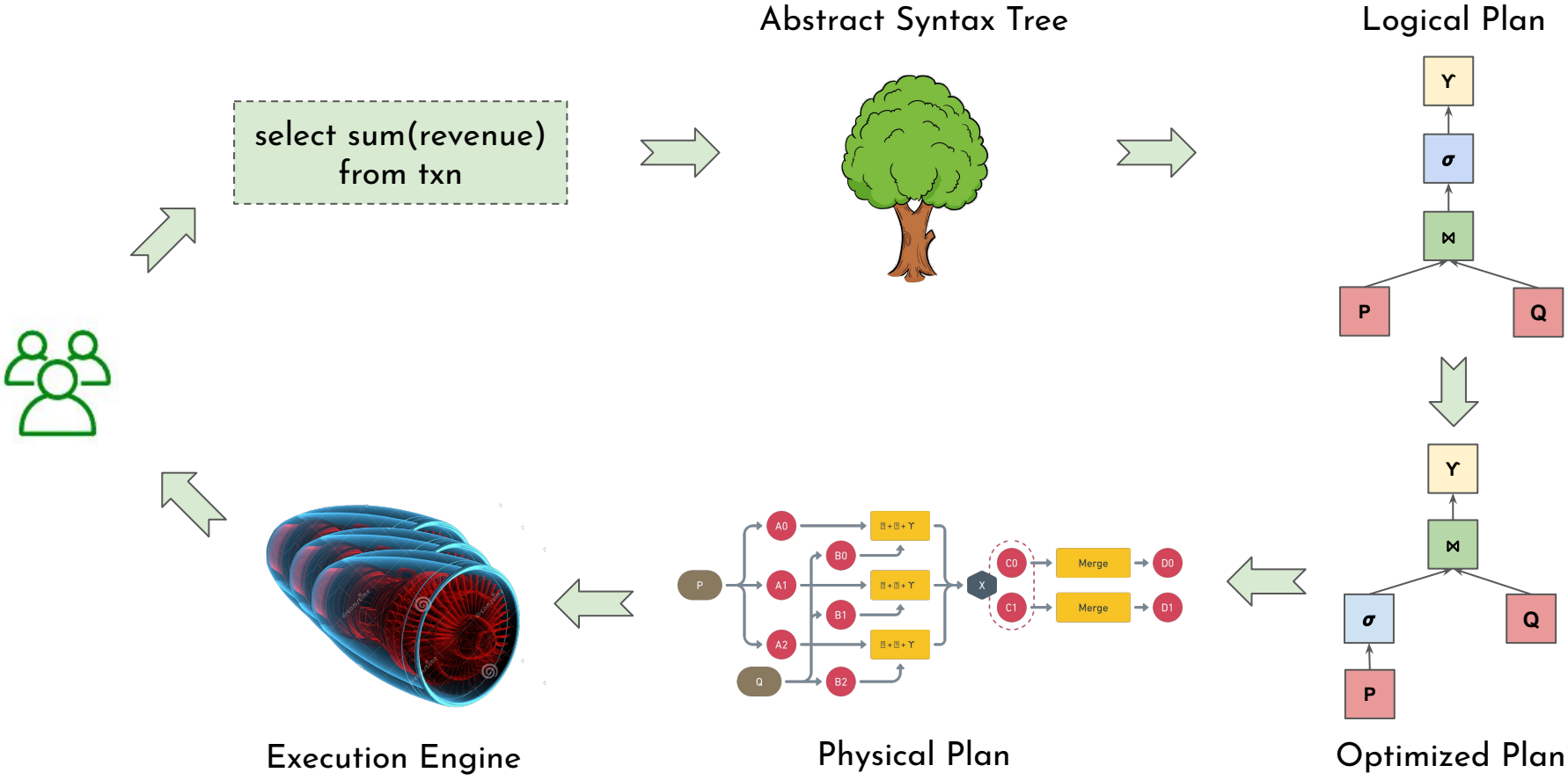
CAP: Converged Analytics Platform



Agenda

- Introduction
- **SQL Execution Primer**
- Why generate code (or vectorize)?
- Code generation in NetSpring
- Practical considerations for generating code

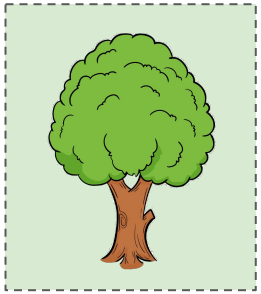
Life Of A Query



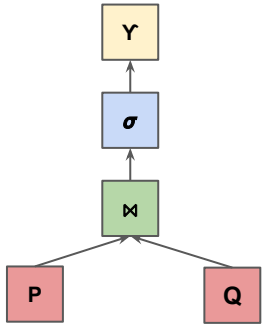
Life Of A Query

select sum(revenue)
from txn

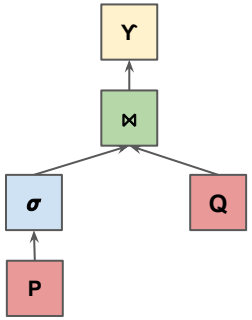
Abstract Syntax Tree



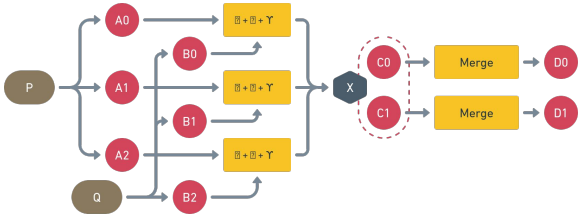
Logical Plan



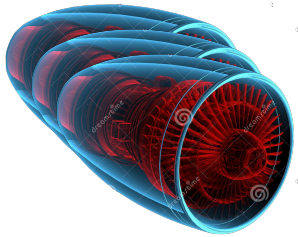
Optimized Plan



Physical Plan



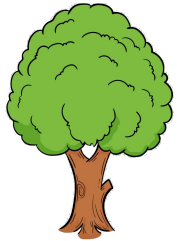
Execution Engine



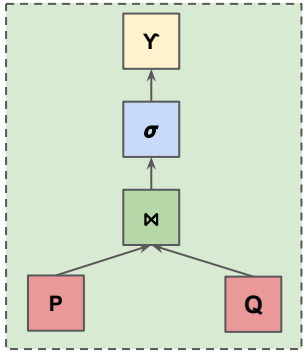
Life Of A Query

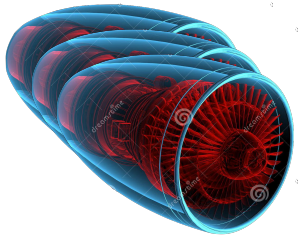
select sum(revenue)
from txn

Abstract Syntax Tree

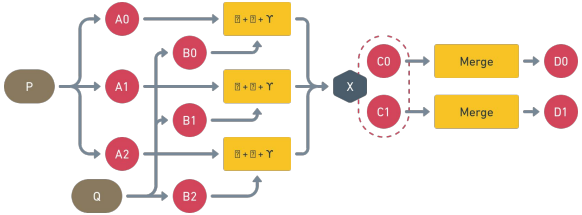


Logical Plan

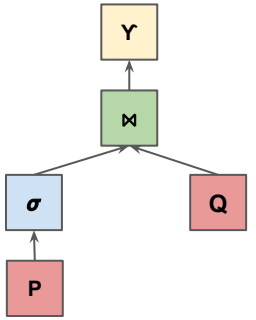




Execution Engine



Physical Plan

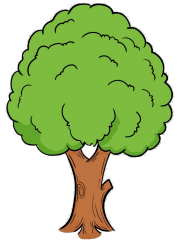


Optimized Plan

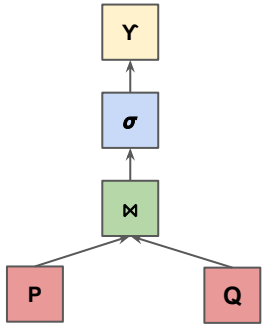
Life Of A Query

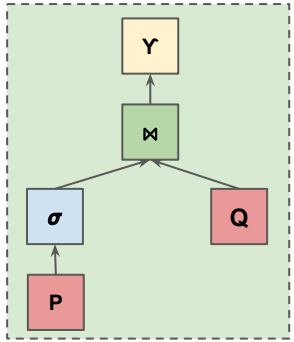
select sum(revenue)
from txn

Abstract Syntax Tree

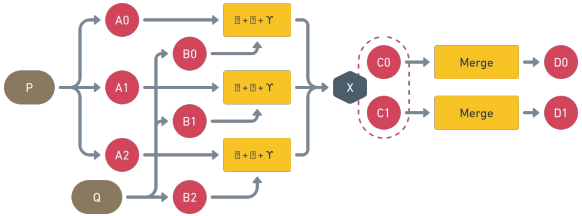


Logical Plan

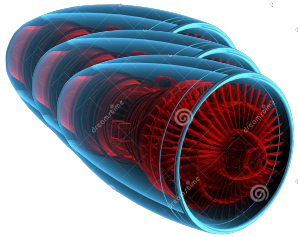




Optimized Plan

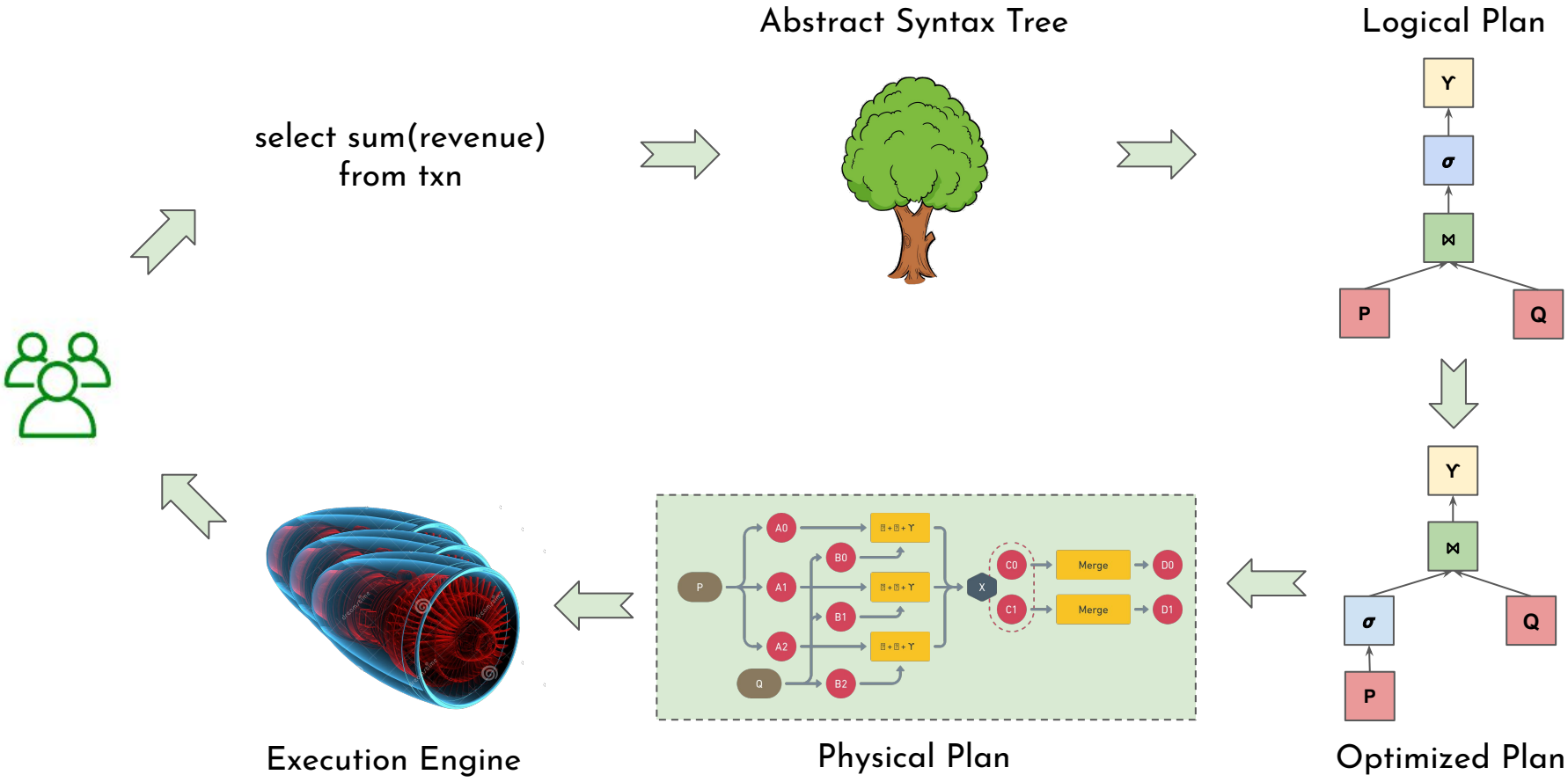


Physical Plan



Execution Engine

Life Of A Query

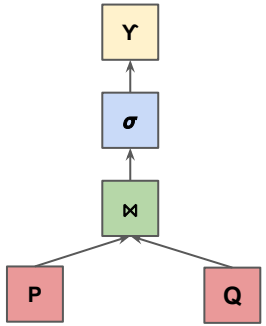


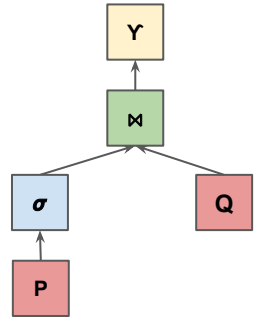
Life Of A Query

select sum(revenue)
from txn

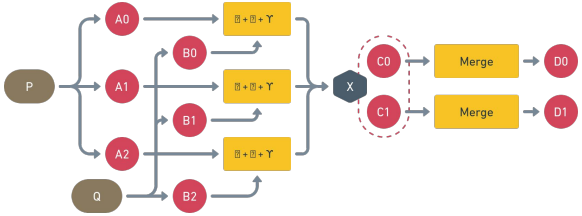
Abstract Syntax Tree

Logical Plan

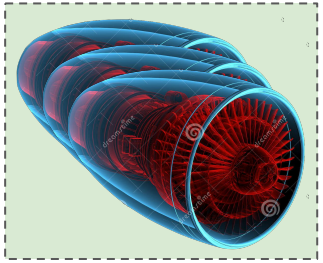




Optimized Plan



Physical Plan



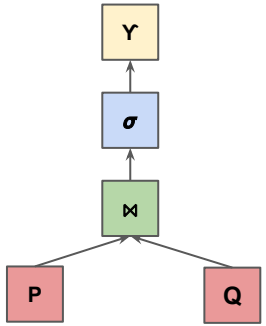
Execution Engine

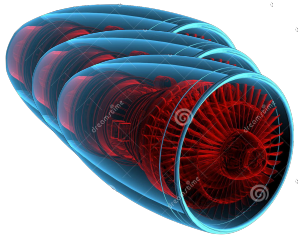
Life Of A Query

select sum(revenue)
from txn

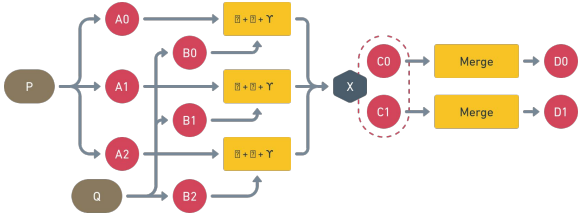
Abstract Syntax Tree

Logical Plan

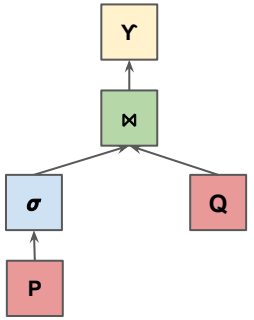




Execution Engine

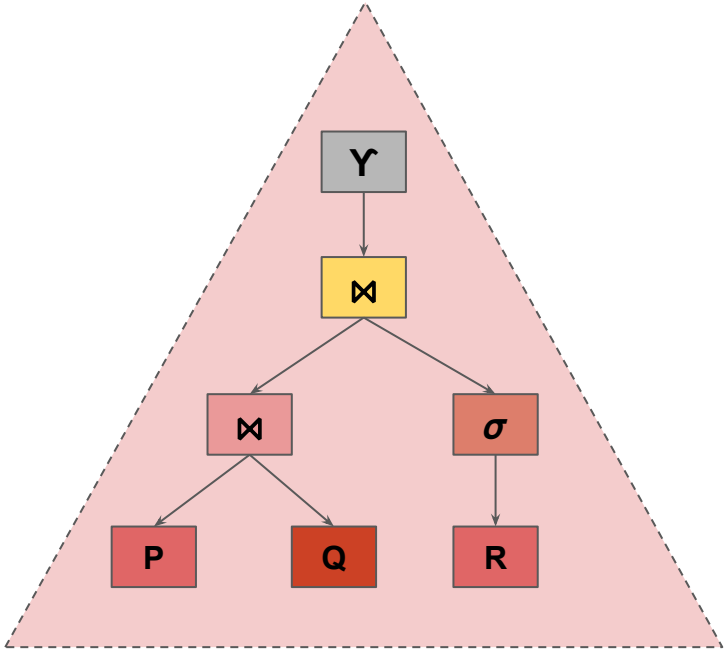


Physical Plan



Optimized Plan

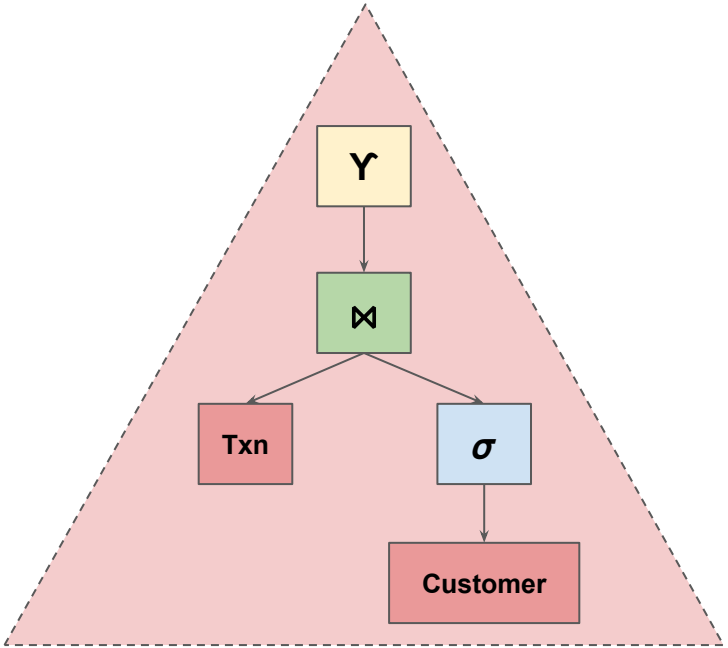
A Volcano of Operators



Volcano - An Extensible and Parallel Query Evaluation System [1]

Iterator Interface

select sum(revenue) from txn join customer on txn.cust_id = customer.id
where customer.city = 'san francisco'




```
class Iterator {
public:
  virtual ~Iterator() = default;
  // Initialize the iterator
  virtual void open() = 0;
  // Read next output row
  virtual Row next() = 0;
  // Release all allocated resources
  virtual void close() = 0;
};
```

Agenda

- Introduction
- SQL Execution Primer
- **Why generate code (or vectorize)?**
- Code generation in NetSpring
- Practical considerations for generating code

Bottleneck has moved to the CPU

Cheaper Memory

Faster Disks

Elastic Disk Bandwidth

Cost of Abstraction

```
class Iterator {
public:
  virtual ~Iterator() = default;
  // Initialize the iterator
  virtual void open() = 0;
  // Read next output row
  virtual Row next() = 0;
 // Release all allocated resources
  virtual void close() = 0;
};
```

Cost of Abstraction: Dynamic Dispatch

```
class Iterator {
public:
  virtual ~Iterator() = default;
  // Initialize the iterator
  virtual void open() = 0;
  // Read next output row
  virtual Row next() = 0;
 // Release all allocated resources
  virtual void close() = 0;
};
```

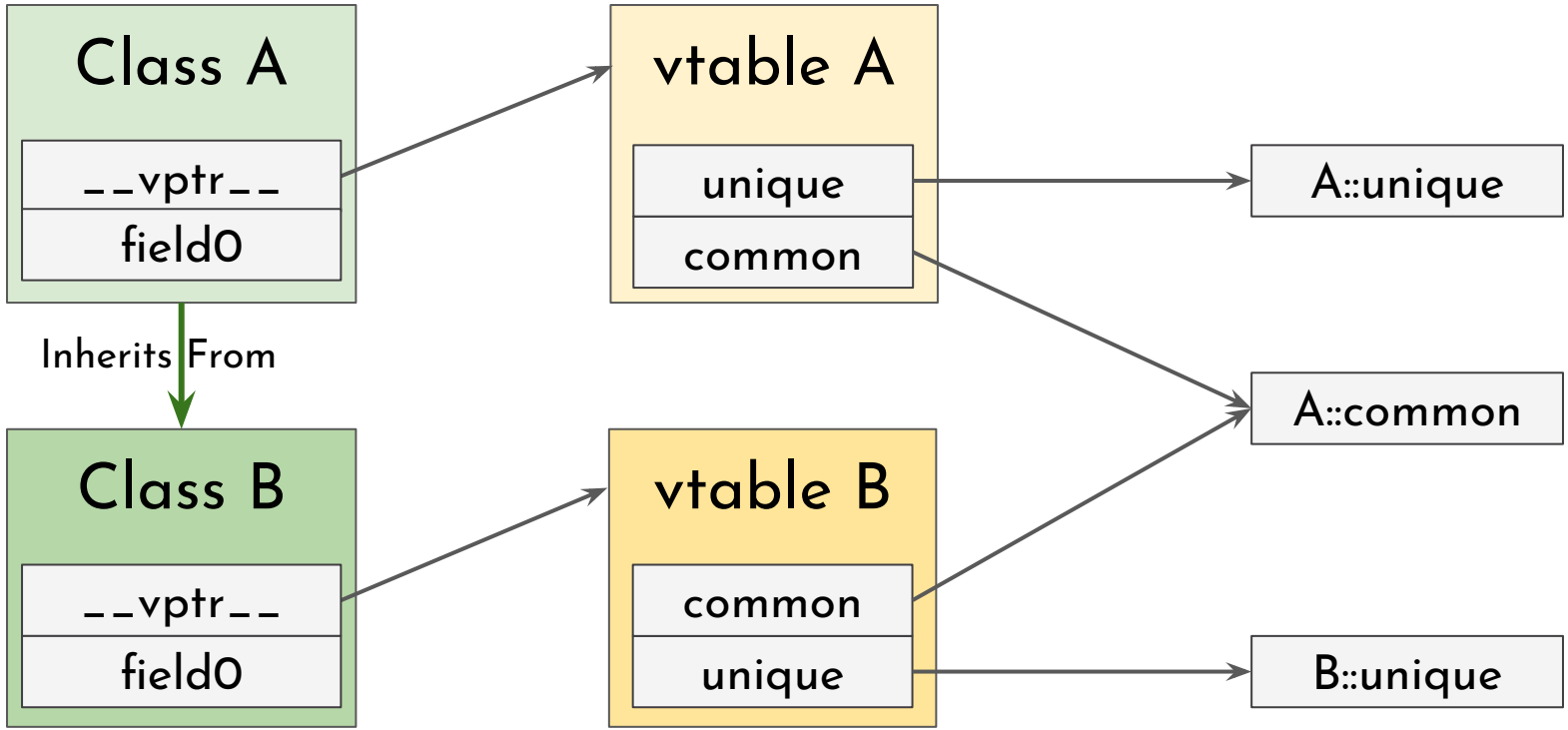
Cost of Abstraction: Union Types

Dynamic Dispatch

```
class Iterator {
public:
  virtual ~Iterator() = default;
  // Initialize the iterator
  virtual void open() = 0;
  // Read next output row
  virtual Row next() = 0;
 // Release all allocated resources
  virtual void close() = 0;
};
```

Union Type

Cost of Abstraction: Dynamic Dispatch



Cost of Abstraction: Union Types

```
// Union struct for all supported types
struct ValueType {
  union {
    int64_t i;
    double d;
    char* s;
    bool b;
    // Other types
  };
  // Metadata for a value.
  Type type;
  bool null;
};

// A row is a list of values.
struct Row { std::vector<ValueType> values; };
```

Performance: select sum(revenue) from txn

Volcano Interface

```
class ScanIterator : public Iterator {
  virtual Row next() {
    if (done()) return emptyRow();
    return mkRow(revenue[offset_++]);
  }
  std::vector<int64_t> revenue_;
  int offset_ = 0;
};
class AggregateIterator : public Iterator {
  virtual Row next() {
    int sum = 0;
    while (true) {
      auto& row = in_->next();
      if (row.values.empty()) break;
      if (row.values[0].type == ValueType::int32)
        sum += in.values[0].i32();
    }
    return mkRow(sum);
  }
  Iterator* in_;
};
```

Hand-written code

Simple is fast

```
int64_t sum = 0;
std::vector<int64_t> revenue(10e6, 1);
for (int i = 0; i < revenue.size(); ++i) {
  sum += revenue[i];
}
return sum;
```

100x fast

Code Generation & Vectorization

Vectorization - VectorWise, Snowflake

Batch tuples across the interface to amortize the cost of interface

- Hides cache-miss latency [3]
- Well suited for SIMD ops

- Non-intuitive algorithms
- Potential to go faster

Code Generation - HyPer, Spark

Generate code with native types and without virtual functions

- Fast on cache resident data [3]
- Natural algorithms

- Compilation cost
- Debugging & Profiling

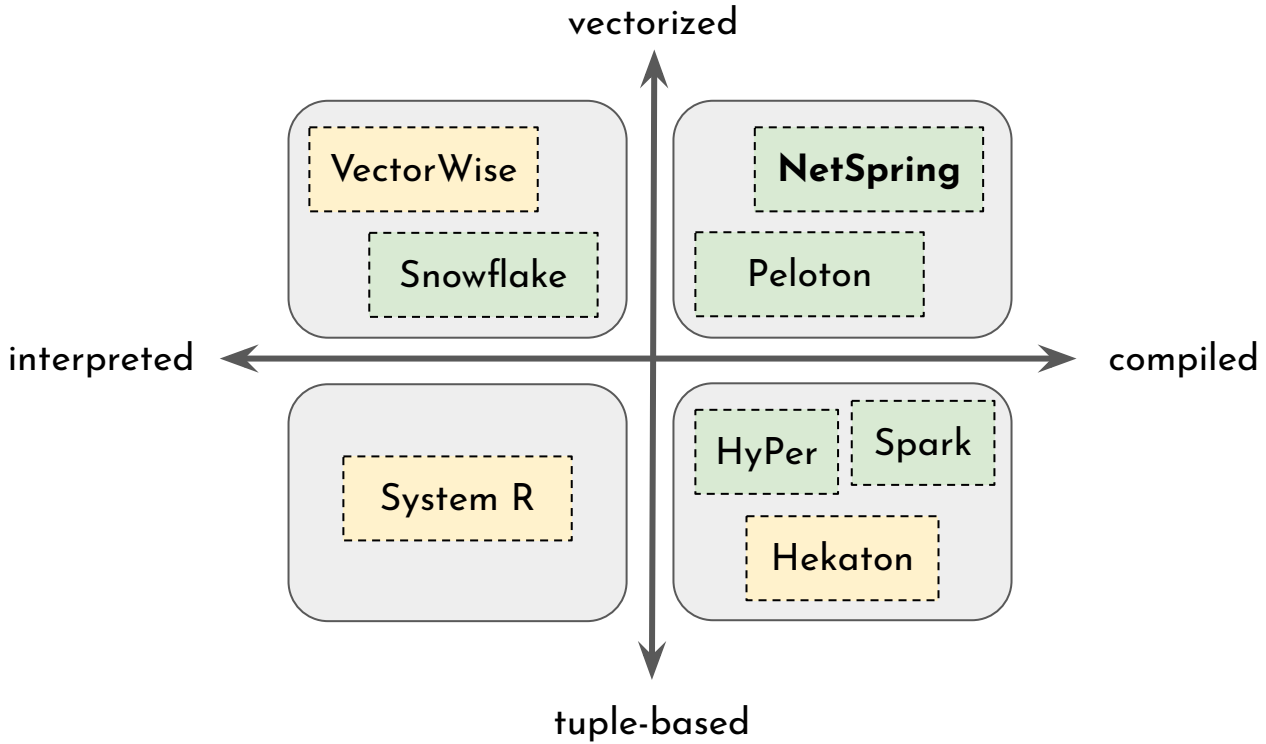
Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask [3]

Agenda

- Introduction
- SQL Execution Primer
- Why generate code (or vectorize)?
- **Code generation in NetSpring**
- Practical considerations for generating code

NetSpring Execution Engine

Push based, vectorized interface with code generation

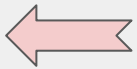


Operator Interface

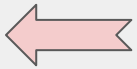
```
class Operator {
public:
  // Returns when ready to consume pkts from @child
  virtual Coro<ns::Status> canConsume(int child);
  // Consumes packets from @child input
  virtual void consume(int child, Packet in);
  // Returns true if the op is ready to produce packets
  virtual bool canProduce();
  // Produces next output packet
  virtual Packet produce();
};
```

Operator Interface

```
class Operator {
public:
  // Returns when ready to consume pkts from @child
  virtual Coro<ns::Status> canConsume(int child);
  // Consumes packets from @child input
  virtual void consume(int child, Packet in);
  // Returns true if the op is ready to produce packets
  virtual bool canProduce();
  // Produces next output packet
  virtual Packet produce();
};
```



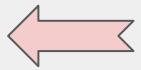
Push Based



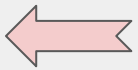
Push Based

Operator Interface

```
class Operator {
public:
  // Returns when ready to consume pkts from @child
  virtual Coro<ns::Status> canConsume(int child);
  // Consumes packets from @child input
  virtual void consume(int child, Packet in);
  // Returns true if the op is ready to produce packets
  virtual bool canProduce();
  // Produces next output packet
  virtual Packet produce();
};
```



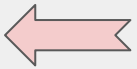
Vectorized



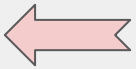
Vectorized

Operator Interface

```
class Operator {
public:
  // Returns when ready to consume pkts from @child
  virtual Coro<ns::Status> canConsume(int child);
  // Consumes packets from @child input
  virtual void consume(int child, Packet in);
  // Returns true if the op is ready to produce packets
  virtual bool canProduce();
  // Produces next output packet
  virtual Packet produce();
};
```



Code Generated



Code Generated

Advantage 1: Faster Execution

CodeGen

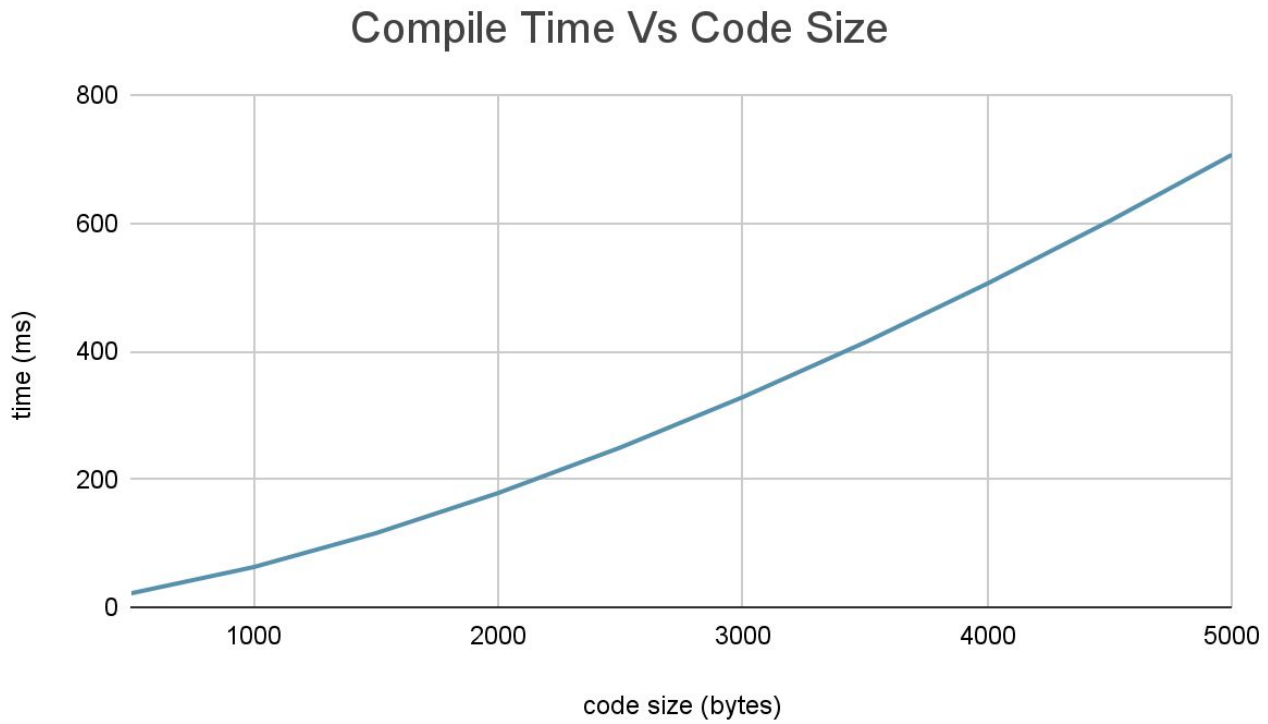
fewer instructions

+

Vectorize

better pre-fetching

Advantage 2: Faster Compilation



Advantage 3: Compile Cache Hit Rate

One compilation per exec node

Duplicate operators share modules

75% for tpc-ds

Re-compiles changed operators only

Advantage 4: Intuitive Algorithms

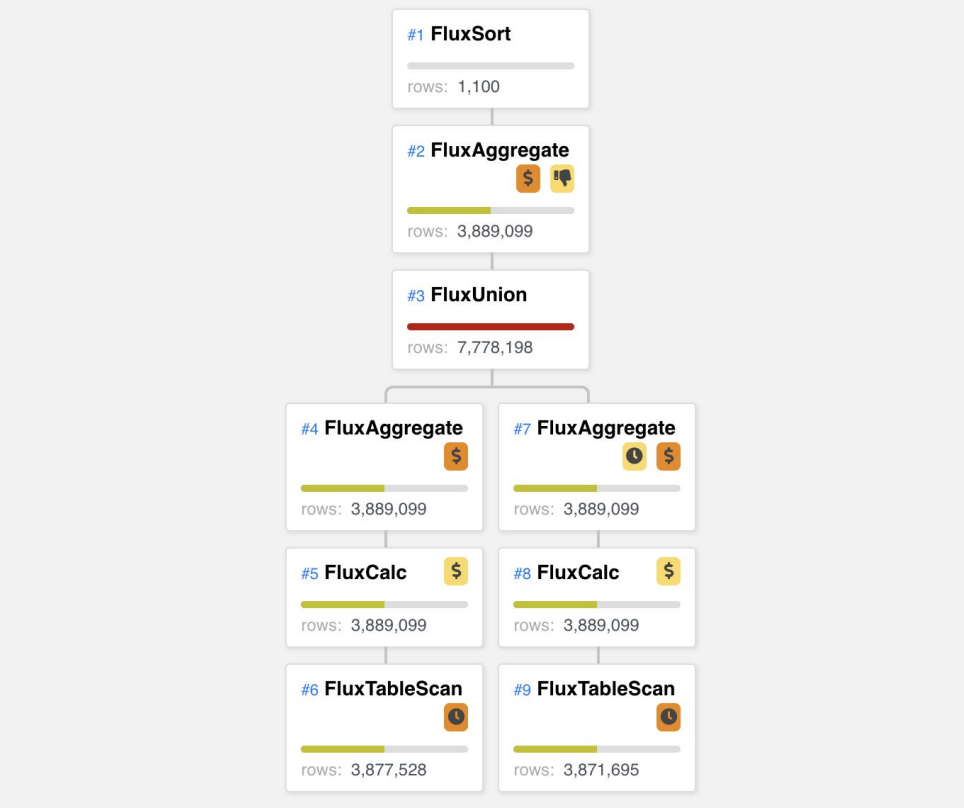
CodeGen Aggregate

```
HashMap<string, int64_t> ht;
for (int r = 0; r < pkt.numRows(); ++r) {
  ht[pkt.col(0).s(r)] += pkt.col(1).i64(r);
}
produce(Packet(ht));
```

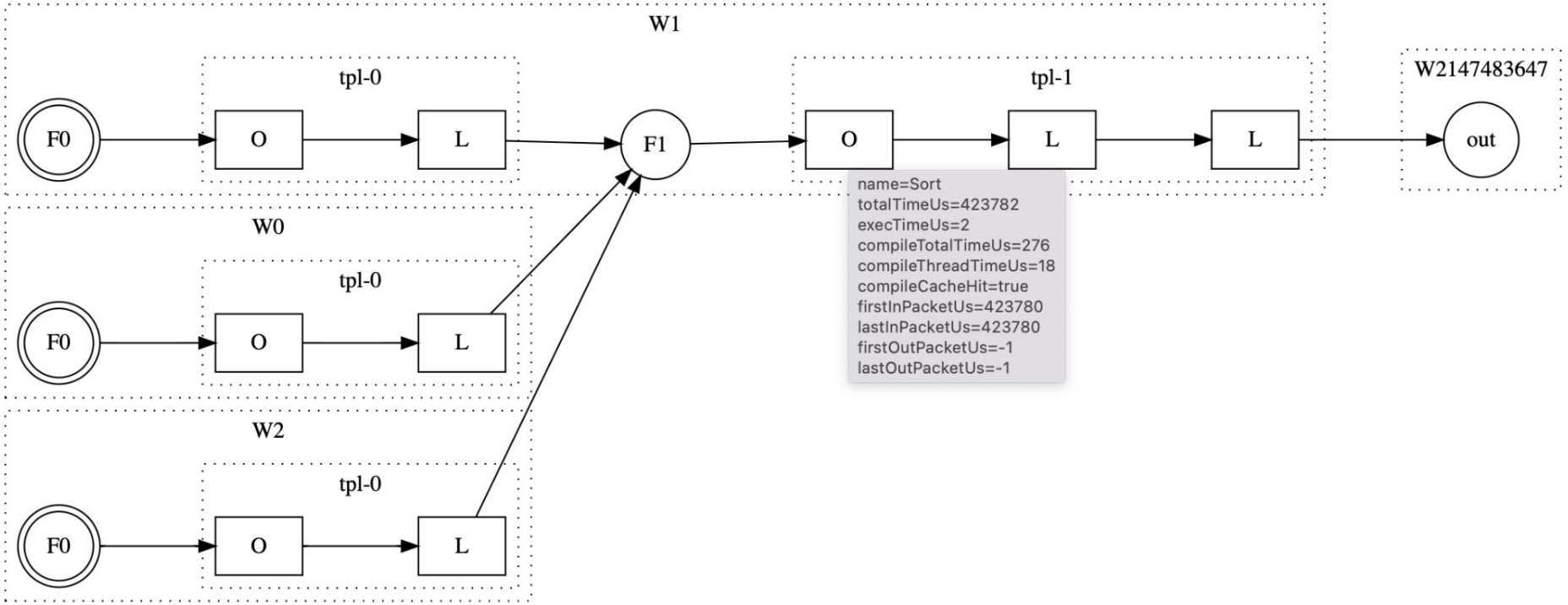
Vectorized Aggregate

```
HashMap<GroupKey, GroupValue> ht;
auto keys = calcKeys(pkt);
auto hashes = calcHashes(keys);
auto entries = ht.findEntries(hashes);
vector<bool> found;
while (entries.size() > 0) {
  DEFER(walkEntryChain());
  auto isEq = keyEqual(pkt, entries);
  addAggregate(pkt, entries, isEq);
  computeOr(isEq, found);
}
auto newEntries =
extractNewEntries(entries, not(found));
ht.findEntries(newEntries);
addAggregate(pkt, newEntries);
```

Advantage 5: Easier Instrumentation



Advantage 5: Easier Instrumentation



Agenda

- Introduction
- SQL Execution Primer
- Why generate code (or vectorize)?
- Code generation in NetSpring
- **Practical considerations for generating code**

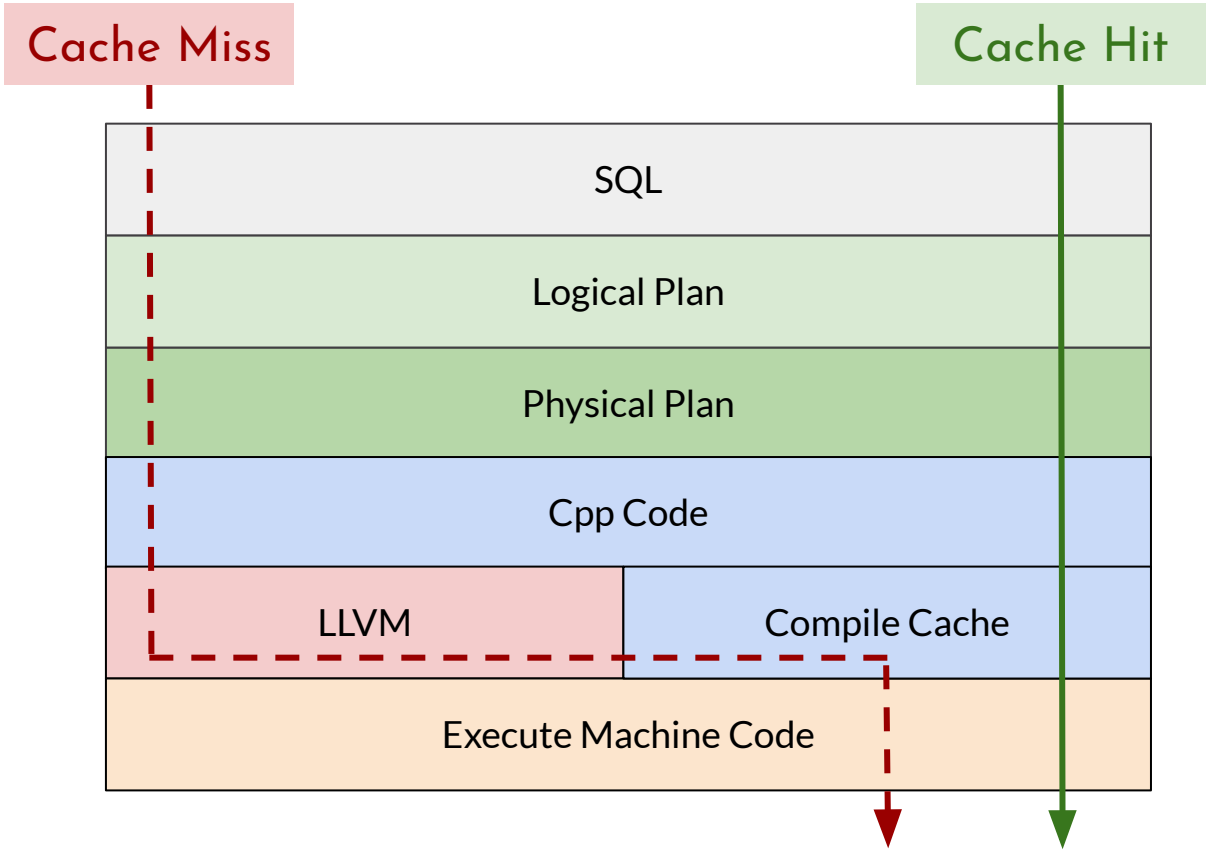
Practical Considerations For Generating Code

Compilation Cost

Difficult To Debug

Pain To Profile

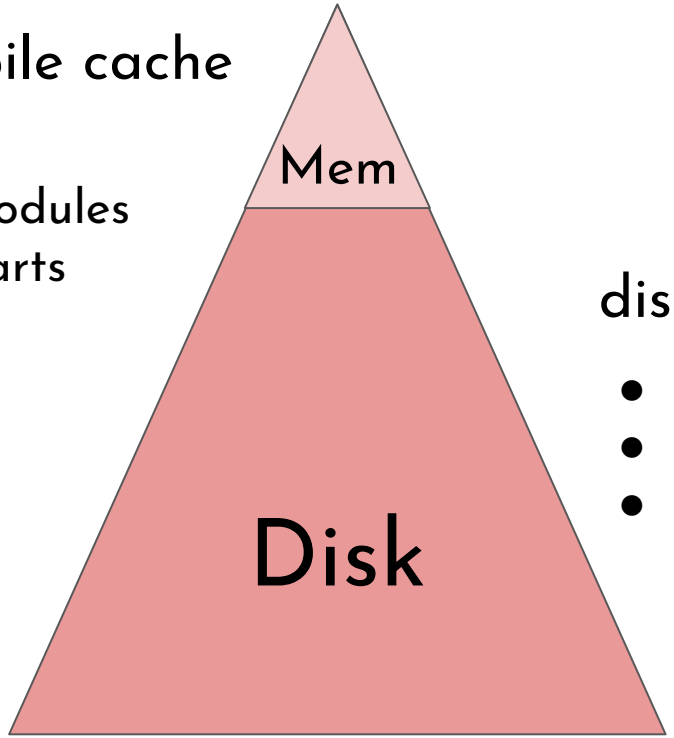
Compilation Cost: LLVM Module Caching



Compilation Cost: Secondary Compile Cache

memory compile cache

- Fast lookup
- Caches 1k modules
- Lost on restarts



disk compile cache

- Serialize LLVM modules to disk
- Practically Infinite Cache
- Survive restarts

Compilation Cost: Parameterized Constants

No constants in generated code

Compilation Cost: Pre-compiled Runtime

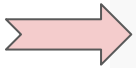
Pre compiled runtime

Symbol Table for Generated Code

JITEventListener to map address ranges to JIT symbols

Difficult to Debug: Symbolize Crash Stacktrace

JIT Addr on heap

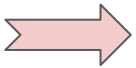


- Keep frame pointers
- Include debug symbols
- Add crash handler

```
*** Received crash signal. See stderr for stacktrace ***
*** JIT Symbolized StackTrace ***
    @ 00000000037f6bb folly::symbolizer::getStackTraceSafe(unsigned long*, unsigned long)
    @ 00000000000e0b1 ns::(anonymous namespace)::onFatalSignal()
    @ 00000000037f451 folly::symbolizer::(anonymous namespace)::signalHandler(int, siginfo_t*, void*)
    @ 000000000000000 (null)
    @ 0007f172f963920 flux::JoinOp::produceData()
    @ 0007f172f964c0a consumeDataLeft
    @ 00000000004809c flux::HashEquiJoinOperator::consumeData(int, flux::Packet)
    @ 00000000000e79c flux::Operator::consume(int, flux::Packet)
    @ 00000000005e882 flux::Turbine::driveOperator(flux::OperatorTree::Node*, flux::Packet) [clone .res
    @ 000000000464e80 folly::resumeCoroutineWithNewAsyncStackRoot(std::experimental::coroutines_v1::cor
    @ 00000000004f9da void folly::detail::function::FunctionTraits<void ()>::callSmall<folly::coro::det
    @ 0000000003f581a folly::EventBase::FunctionLoopCallback::runLoopCallback()
    @ 0000000003f0078 folly::EventBase::runLoopCallbacks()
    @ 0000000003f08cb folly::EventBase::loopBody(int, bool)
    @ 0000000003f36ef folly::EventBase::loopForever()
    @ 0000000000839c5 std::thread::_State_impl<std::thread::_Invoker<std::tuple<flux::FluxServer::start
cutor, std::default_delete<net::FollyExecutor> > > > const&, std::__cxx11::basic_string<char, std::char
    @ 000000000000000 (null)
    @ 000000000009608 start_thread
    @ 000000000122292 clone
```

Difficult to Debug: GDBJITEventListener

JIT Symbols



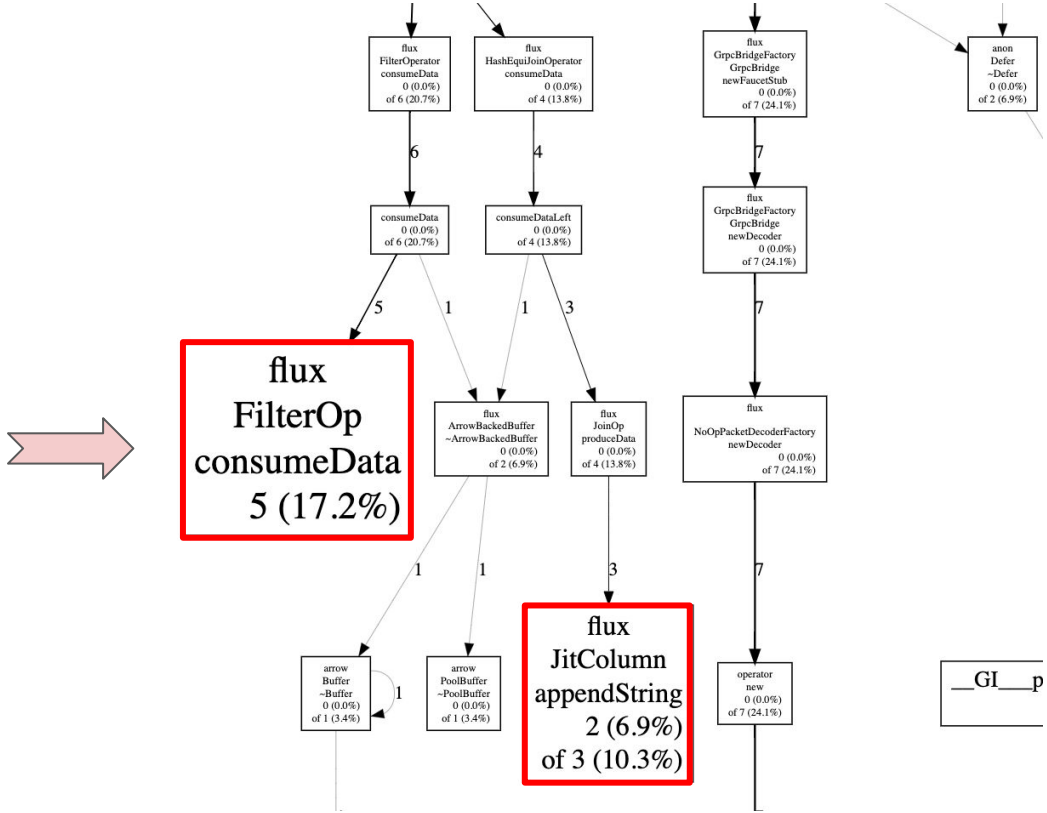
- Provided by LLVM
- Uses GDB JIT interface to communicate symbols
- Turn on while debugging

```
(gdb) bt
#0  0x00007fee56f4d2f0 in produceData () at jitcode.cpp:1620
#1  0x00007fee56f4ec0b in consumeDataLeft () at jitcode.cpp:1574
#2  consumeDataLeft () at jitcode.cpp:1701
#3  0x00007fee565ef0bd in flux::HashEquiJoinOperator::consumeData (thi
#4  0x00007fee55ffd79d in flux::Operator::consume (this=0x7fee3b07d800
#5  0x00007fee5688f883 in flux::Turbine::driveOperator (this=<optimize
#6  0x00007fee4b9d4e81 in std::experimental::coroutines_v1::coroutine_
#7  folly::resumeCoroutineWithNewAsyncStackRoot (h=..., frame=...) at
#8  0x00007fee56b0c9db in folly::coro::detail::ViaCoroutinePromiseBase
```

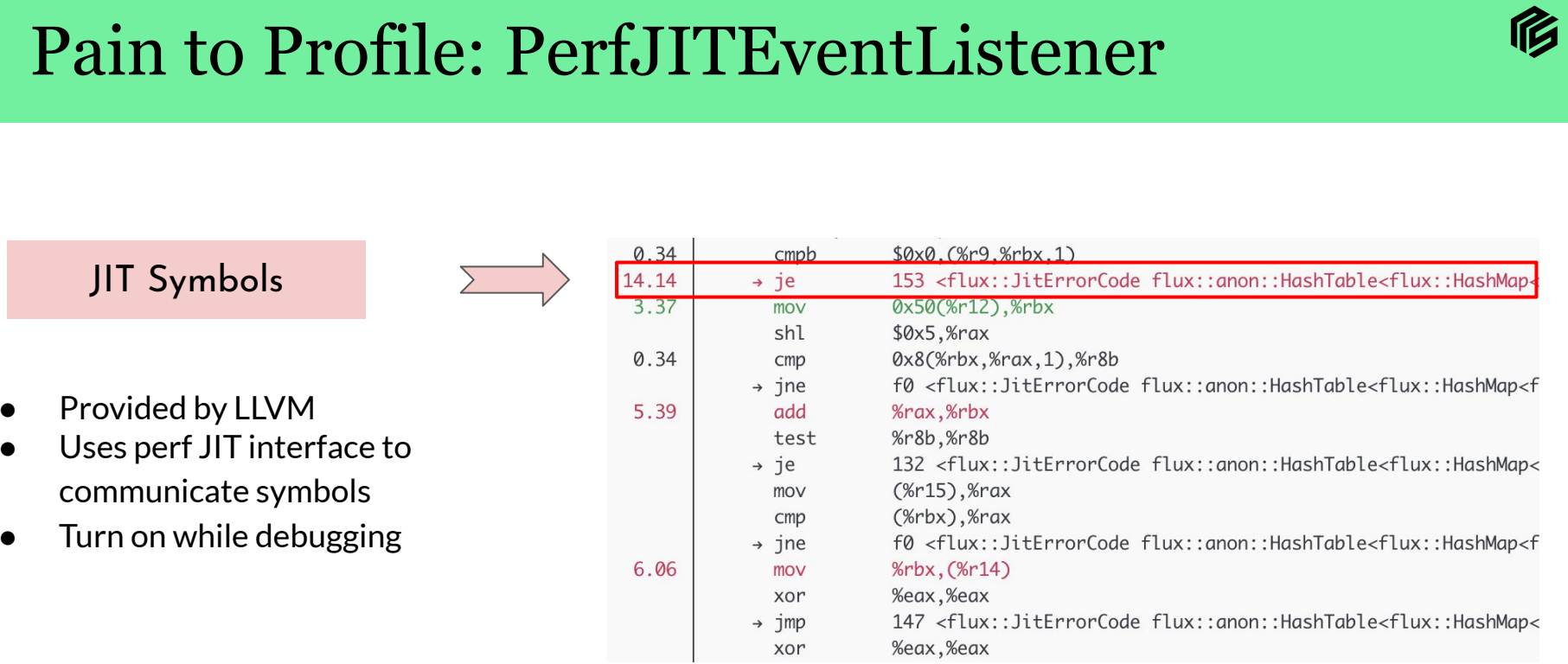
Pain to Profile: Symbolized Gperf Profiles

Generate symbolized profiles with JIT symbols

JIT Symbols



Pain to Profile: PerfJITEventListener



Summary

- Abstractions while powerful have associated costs
- Code generation on vectorized interface brings the best of both worlds
- Debugging JIT code can be configured with some effort

twitter.com/satyamshekhar

satyam@netspring.io

References

[1] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System.

[2] T. Neumann. Efficiently compiling efficient query plans for modern hardware.

[3] P. Menon et al. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last.

[4] Kersten et al. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask.