Building Efficient Query Engines in a High-Level Language

Yannis Klonatos, Christoph Koch, Tiark Rompf, Hassan Chafi

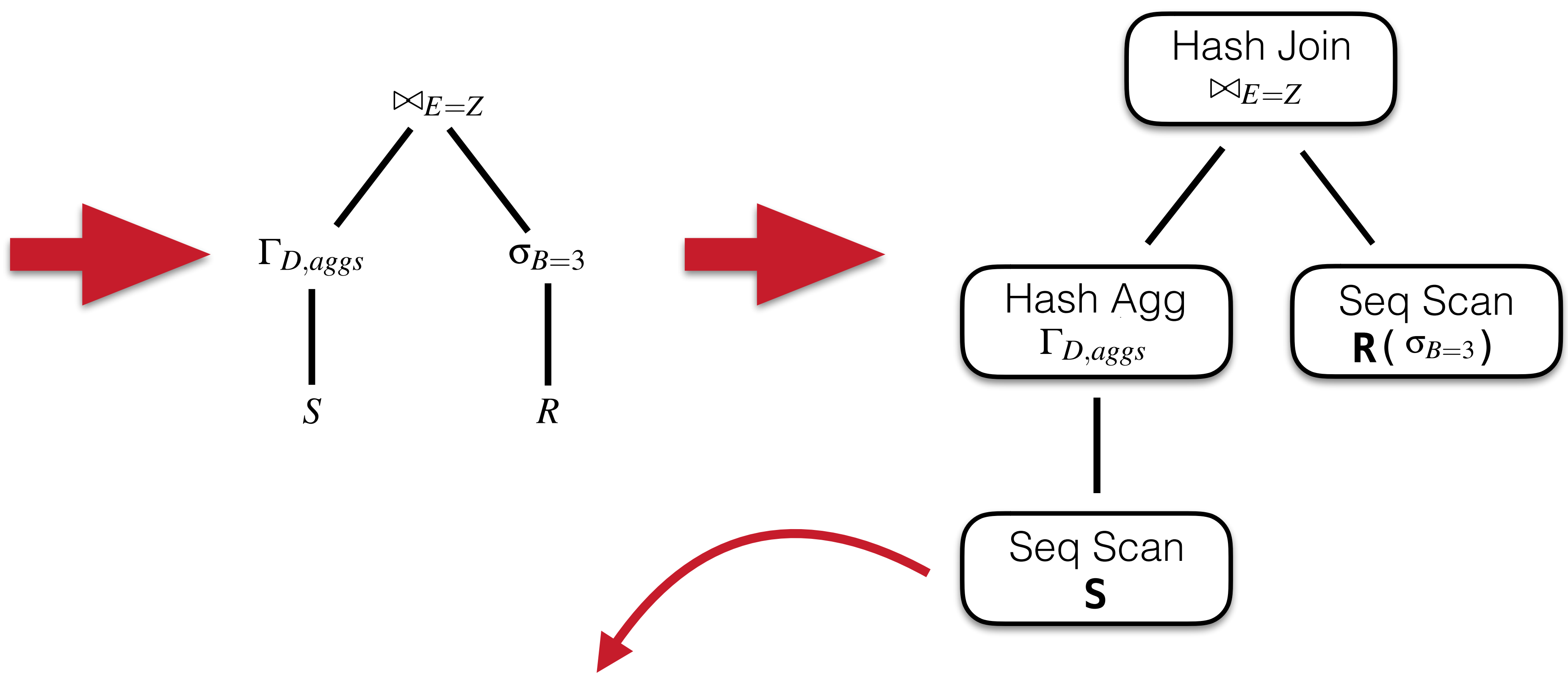
Prashanth Menon, Reading Group Fall 2015

CARNEGIE MELLON
DATABASE GROUP

Background

- What happens to your SQL query?

```
select *
from R, (select S.D,
   sum(1-S.B) as E,
   sum(S.A*(1-S.B)),
   sum(S.A*(1-S.B)*(1+S.C))
  from S group by S.D) T
where R.Z=T.E and R.B=3
```



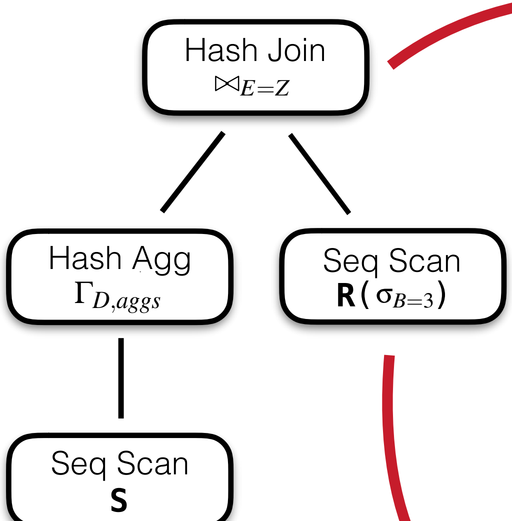

```
class Operator {
  virtual void open() = 0;
  virtual Tuple* next() = 0;
  virtual void close() = 0;
};
```

Background (2)

- Volcano model is powerful, generic and composable
- Designed in an era where disk I/O dominated overhead
- If all data stored in main memory, it doesn't perform well
  - All **`next()`** calls are virtual (i.e., vtable lookup)
  - Single function call overhead for each tuple, for each operator!
  - Pretty poor cache utilization
- Can we do better?

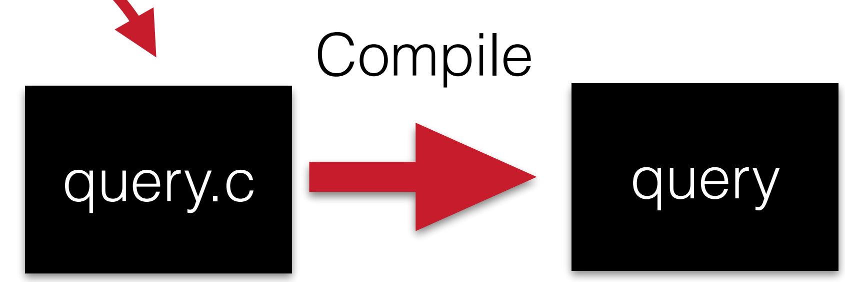
Background (3)

- Generate a per-query execution engine!




```
// loop over pages
for (int p_R = start_page_R; p_R <= end_page_R; p_R++) {
 page_struct *page_R = read_page(p_R, R);
 for (int p_S = start_page_S; p_S <= end_page_S; p_S++) {
  page_struct *page_S = read_page(p_S, S);

  // loop over tuples
  for (int t_R = 0; t_R < page_R->num_tuples; t_R++) {
   void *tuple_R = page_R->data + t_R * tuple_size_R;
    for (int t_S = 0; t_S < page_S->num_tuples; t_S++) {
    void *tuple_S = page_S->data + t_S * tuple_size_S;
    int *attr_R = tuple_R + offset_R;
    int *attr_S = tuple_S + offset_S;
    if (*attr_R != *attr_S) continue;
    add_to_result(tuple_R, tuple_S);     /* inlined */
}}}}
```




```
// loop over pages
for (int p = start_page; p <= end_page; p++) {
 page_struct *page = read_page(p, table);
 // loop over tuples
 for (int t = 0; t < page->num_tuples; t++) {
  void *tuple = page->data + t * tuple_size;
  int *value = tuple + predicate_offset;
  if (*value != predicate_value) continue;
  memcpy(..);
}}
```

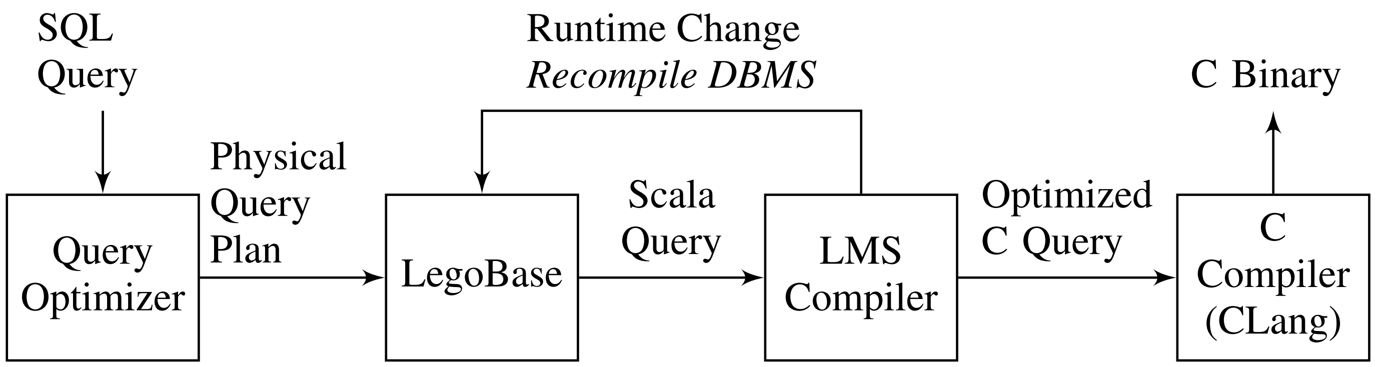
Background

- Compiling queries will yield better performance
- However, template expansion is:
  - Brittle
  - Very low level (i.e., hard to implement)
  - Limited scope of compilation
  - Limited adaptivity

Goal

- Performance of low-level hand-written query code
- Productivity of high-level language with rich type system guarantees

LegoBase



- Query engine written in Scala
- Cross-compiles Scala query plans into optimized C code
- Four steps:
    1. Convert pre-assembled physical query plan to naive Scala-based operator tree
    2. Use Lightweight Modular Staging (LMS) to convert operator tree into Scala IR
    3. Execute multiple optimization passes on IR
    4. Output optimized Scala or C query plan
- Optimizations are written in Scala, operate on Scala types
- **Programmatic removal of abstraction overhead**

Optimizations

- Optimizations are performed in LMS passes
  - Similar to LLVM where passes are independent
- Optimizations include:
  - Inter-operator optimizations
  - Eliminating redundant materializations
  - Data structure specialization
  - Data layout changes
  - Traditional compiler optimizations (DCE, loop unrolling)

Inter-Operator Optimizations

- Convert query plan from pull-based to push-based (à la HyPer)
  - Operators **push** data to consumer operators
  - Better cache locality (no function calls, tuples remain in registers)

```scala
case class HashJoin[B](leftChild: Operator,
    rightChild: Operator, hash: Record=>B,
    cond: (Record,Record)=>Boolean) extends Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  var it: Iterator[Record] = null
  def next() : Record = {
    var t: Record = null
    if (it == null || !it.hasNext) {
      t = rightChild.findFirst { e =>
        hm.get(hash(e)) match {
          case Some(hl) => it = hl.iterator; true
          case None => it = null; false
        }
      }
    }
    if (it == null || !it.hasNext) return null
    else return it.collectFirst {
      case e if cond(e,t) => conc(e, t)
    } get
  }
}
```

```scala
case class HashJoin[B](leftChild: Operator,
    rightChild: Operator, hash: Record=>B,
    cond: (Record,Record)=>Boolean) extends Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  var it: Iterator[Record] = null
  def next(t: Record) {
    var res: Record = null
    while (res = {
      if (it == null || !it.hasNext) {
        hm.get(hash(t)) match {
          case Some(hl) => it = hl.iterator
          case None => it = null
        }
      }
      if (it == null || !it.hasNext) null
      else it.collectFirst {
        case e if cond(e,t) => conc(e, t)
      } get
    } != null) parent.next(res)
  }
}
```

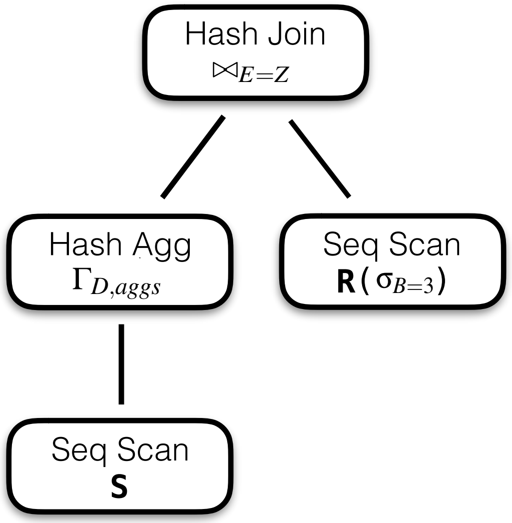
Inter-Operator Optimizations (2)

- Convert query plan from pull-based to push-based (à la HyPer)
  - Operators **push** data to consumer operators
  - Better cache locality (no function calls, tuples remain in registers)

```scala
case class HashJoin[B](leftChild: Operator,
   rightChild: Operator, hash: Record=>B,
   cond: (Record,Record)=>Boolean) extends Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  var it: Iterator[Record] = null
  def next(t: Record) {
    if (it == null || !it.hasNext) {
      hm.get(hash(t)) match {
        case Some(hl) => it = hl.iterator
        case None => it = null
      }
    }
    while (it!=null && it.hasNext) it.collectFirst {
      case e if cond(e,t) => parent.next(conc(e,t))
    }
  }
}
```

```scala
case class HashJoin[B](leftChild: Operator,
   rightChild: Operator, hash: Record=>B,
   cond: (Record,Record)=>Boolean) extends Operator {
  val hm = HashMap[B,ArrayBuffer[Record]]()
  def next(t: Record) {
    hm.get(hash(t)) match {
      case Some(hl) => hl.foreach { e =>
        if (cond(e,t)) parent.next(conc(e,t))
      }
      case None => {}
    }
  }
}
```

Redundant Materialization



- Not necessary to materialize aggregations
- Can bypass aggregation node and perform aggregation in build phase of join
- Difficult (probably not impossible) to express when using code templates

Redundant Materialization

- Implemented as IR pass
- If we see an HJ node whose left child is Agg grouping on same join attribute, merge them

```
def optimize(op: Operator): Operator = op match {
  case hj@HashJoin(aggOp:AggOp,_,h,eq) =>
    new HashJoin(aggOp.child,hj.rightChild,h,eq) {
      override def open() {
        // leftChild is now the child of aggOp
        leftChild foreach { t =>
          val key = hj.leftHash(aggOp.grp(t))
          // Get aggregations from hash map of HJ
          val aggs = hm.getOrElseUpdate(key,
                         new Array[Double](aggOp.aggFuncs.size))
          aggOp.processAggs(aggs,t)
        }
      }
    }
  case x: Operator =>
    x.leftChild = optimize(x.leftChild)
    x.rightChild = optimize(x.rightChild)
  case null => null
}
```

Data-Structure Specialization

- Use schema and query knowledge to specialize hash maps
  - Remove abstraction overhead of generic hash maps
- Three main problems:
  1. Redundant data storage (key is usually subset of value)
  2. Lookups require virtual calls to hashing functions
  3. Hash maps require resizing during runtime
- LegoBase solutions:
  1. Convert hash map to contiguous array (of buckets)
  2. Only store values in nodes
  3. Inline hash and equality functions
  4. Use runtime statistics to predict and allocate size of hash map at compile time

Changing Data Layout

- Possible to switch between row and column form at runtime
  - Does not require rewriting query engine
- Implemented as an IR optimization pass
  - Triggered when we see Array[Record] (array of type record) in IR
  - Possible to implement any new data storage layout as an IR optimization

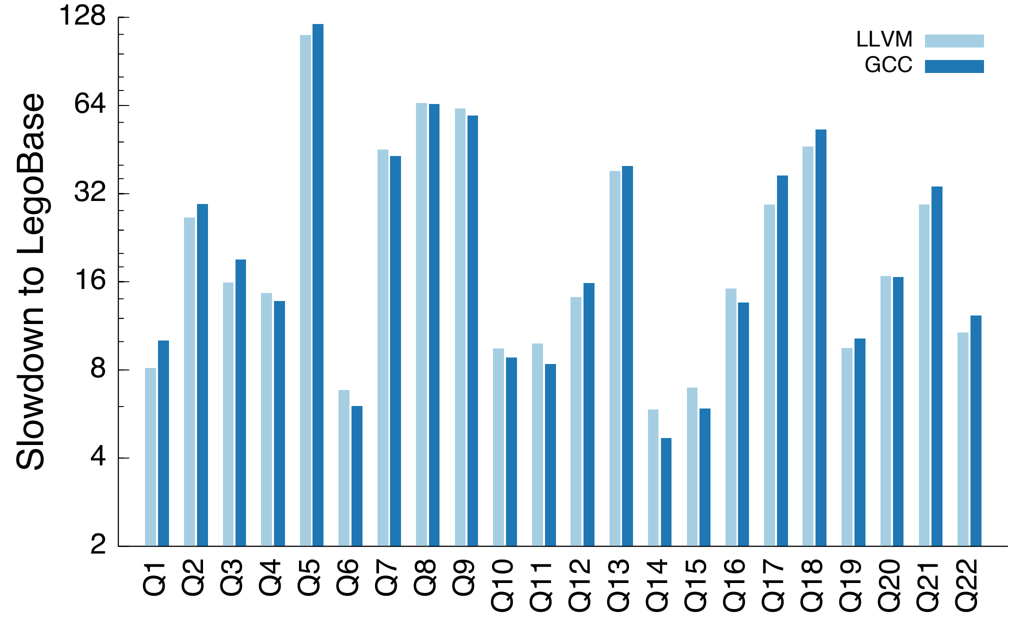
```
override def array_new[T:Manifest](n:Int) =
  manifest[T] match {
    case Record(attrs) =>
      // Create a new array for each attribute
      val arrays = for (tp<-attrs) yield array_new(n)(tp)
      // Pack everything in a new record
      record(attrs, arrays)
    case _ => super.array_new(n)
  }
```

```
override def array_apply[T:Manifest](ar:Array[T],
                                     n:Int) =
  manifest[T] match {
    case Record(attrs) =>
      val arrays = for (l <- attrs) yield field(ar, l)
      val elems = for (a <- arrays) yield a(n)
      // Perform record reconstruction
      record(attrs, elems)
    case _ => super.array_apply(ar, n)
  }
```

Evaluation Setup

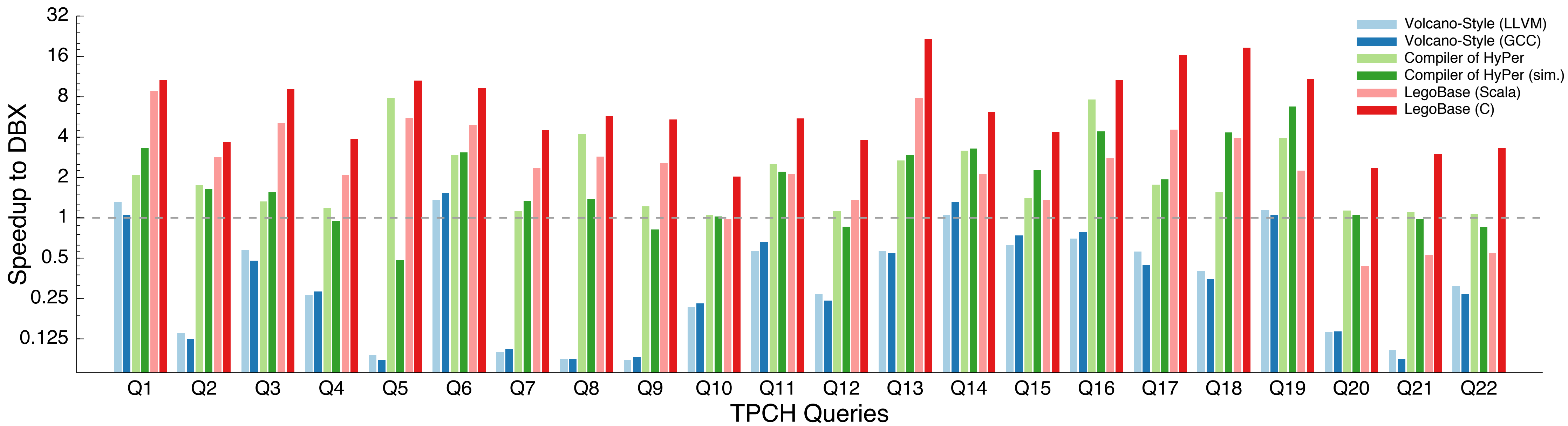
- 2x Intel Xeon 2GHz, 256GB RAM, 2TB HDD
- Scala 2.10.3, Clang 2.9
- Evaluate against DBX (in-memory row-store) and HyPer
- All systems get 192 GB RAM
- Run TPCH

Optimizing Query Plans



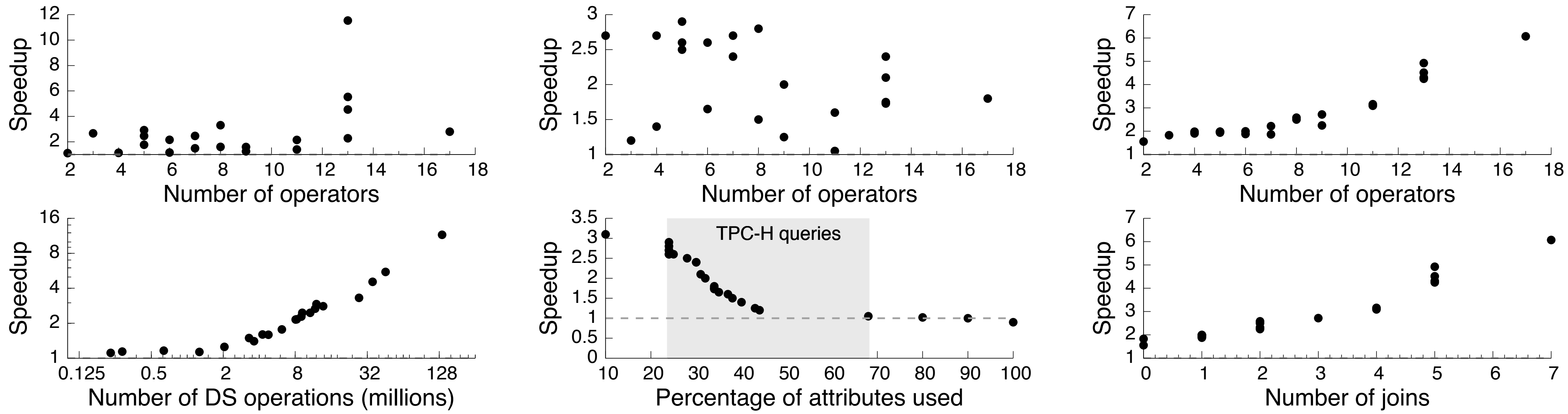
- LegoBase Volcano-style query engine compiled to C
- Compare code compiled with GCC and LLVM against fully optimized LegoBase
- Not all that interesting … really a comparison of GCC and LLVM

TPCH Query Optimization

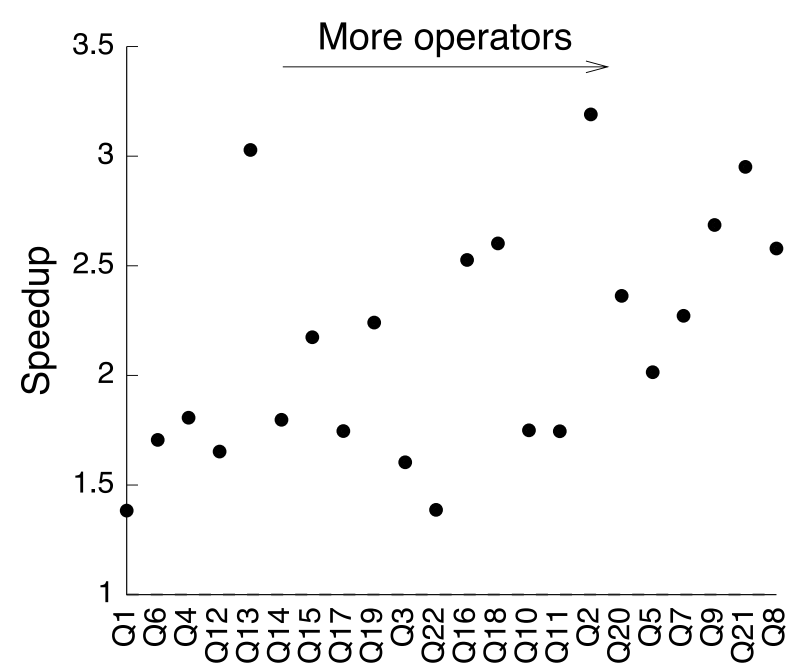


- Simulated HyPer is faster than HyPer
  - Due to data-structure specialization
- LegoBase is 5.3x-7.7x faster than HyPer
  - Due to data-structure specialization, data layout optimization
  - Better cache locality, branch prediction, fewer instructions executed
- LegoBase Scala 2.5x slower than LegoBase C
  - 1.3x-1.4x more branch mispredictions
  - 1.1x-1.8x more LLC misses
  - 5.5x more CPU instructions

Impact of Compiler Optimizations



(a) Data Structure Opt. (b) Change Data Layout (c) Operator Inlining



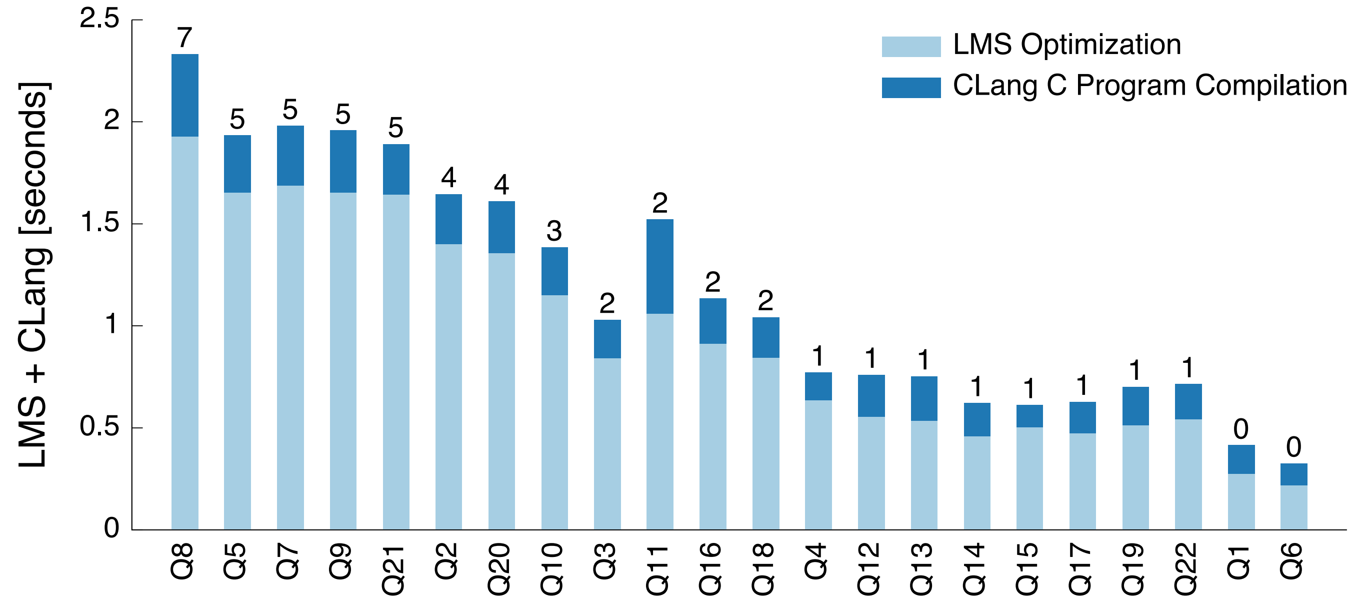
Productivity

	Coding Effort	Scala LOC	Average Speedup
Operator Inlining	–	0	$2.07\times$
Push Engine Opt.	1 Week	$\sim$400 [6]	$2.26\times$
Data Structure Opt.	4 Days	259	$2.16\times$
Change Data Layout	3 Days	102	$1.81\times$
Other Misc. Opt.	3 Days	124	–[10]
LegoBase Operators	1 Month	428	–
LMS Modifications	2 Months	3953	–
Various Utilities	1 Week	538	–
Total	$\sim$4 Months	5831	$7.7\times$

Table 1: Programming effort required for each LegoBase component along with the average speedup obtained from using it.

- Optimizations all done in a high-level language
- Easier to program, fewer lines of code
- High speedup-per-line-of-code

Compilation Overhead



- Compilation time ~ 2.5 seconds

Conclusions

- Possible to build query engine in high-level language with performance of hand-written low-level C
- Use LMS to transform naive query engine to IR
  - Optimize IR in independent stages
  - Specialize types, change data layouts at runtime
  - Emit optimize C code
- Performance beats existing main-memory DBMS and modern query compiler HyPer