

Abstraction without regret in database systems construction

Christoph Koch

EPFL DATA Lab

christoph.koch@epfl.ch

Joint work with

Amir Shaikhha, Yannis Klonatos;

Lewis Brown, Florian Chlan, Thierry Coppey, Mohammad Dashti,

Vojin Jovanovich, Nikos Kokolakis, Andres Noetzli, Martin Odersky (EPFL);

Tiark Rompf (Purdue University)

Abstraction without regret

- Abstraction: develop in modern high-level PL.
 - Make use of generic programming and advanced features for software composition.
- Don't pay the price: get performance competitive with mature, low-level, expert implementation.

Vision:

- **Smart systems compilers replace expert coders.**

CK: “Abstraction Without Regret in Database Systems Building: a Manifesto”. IEEE Data Eng. Bull. 37(1): 70-79 (2014)

DBMS are hard to build

- $\sim 10^7$ lines of C code
 - Modern high-level PL can give up to 100x productivity improvements.
 - Anecdotal evidence from enthusiasts?
- Monolithic.
 - Abstraction leakage: log sequence numbers, page abstraction, etc.
 - Storage Mgr, Buffer Mgr, CC, recovery, query operators all become one monolithic code magma!
 - Modern PL features (genericity, mixin composition, OO features to support design patterns) can fix this.
 - But can we make them fast enough?

The case for query compilers

- “Naïve mainstream”: Interpretation of relational algebra plans. Slow.
- Compilation
 1. reduces #instructions executed at runtime
 2. Improves cache-locality
- Interpretation on steroids: Volcano-style, vectorization, ...
 - Compilation is still faster. Simpler.
 - Volcano-style improves out-of-core data locality at the cost of cache-locality.
 - Vectorization only approaches the performance of compiled code at its sweet spot (vector size).

Query compilers vs. systems compilers

1. Query compilers – part of a DBMS; an implementation “trick”.
2. Classical compilers (eg. C) – do not replace low-level coding.
3. Systems compiler: compiler that “has the skills of a systems programming experts”.
(Separate from the system.)
 - But is such a compiler feasible?
4. Extensible DSL compiler platforms.
 - Make behavior and performance optimizations orthogonal.
 - Compiler (optimization) – system co-design

Implementation
of system in high-
level PL

DS optimizations



Compiler

- Create library of optimizations. With time, converge towards (3).
- Create library of composable systems components.

Compiler optimizations for systems

- Fusion & deforestation
 - `myCollection.map(f).map(g) => myCollection.map(f; g)`
- Row to column transform: important in main mem too!
 - `list<pair<S, T>> => pair<list<S>, list<T> >`
- Data structure specialization
 - e.g. dynamic to static array (based on schema or JIT).
- Fusion, deforestation, code motion and data structure specialization for advanced data structures is hard!
 - Good data structure implementations are imperative and low-level. Hard for automatic program analysis!
- Mainstream compilers typically can't do this.
 - Conservative with optimization features.

Can compilers match/beat human systems programming experts ?

- Empirical evidence in this talk: few techniques account for most of systems performance.
 - Can automatize these in a compiler.
- Just-in-time compilation can eliminate abstraction overheads that can't be eliminated at development time.
 - Schema/data dict abstraction; System-app interface.
- Even in mature code bases: code quality varies.
 - Humans don't have the stamina of a compiler.
 - Some optimizations would (temporarily) take code bases into the 10^9 lines-of-code range: beyond humans
 - Example: inlining (next slide)

The Inlining Software Crisis

- PostgreSQL:
 - 7 implementations of B-trees,
 - 20 implementations of page abstraction.
- Manually inlined with other code for performance.
 - 8 b-trees, 21 page impls would be faster!
 - Software crisis
- A compiler can inline more aggressively and in a more principled way than a human.

Structure of the remainder of the talk

Modern compiler ideas

- Template expansion vs. step-wise lowering
- Compiler-system co-design
- The SC compiler

Two systems built with a modern DSL compiler

1. LegoBase: OLAP
2. S-Store: OLTP

Step-wise lowering

With

- Yannis Klonatos, Amir Shaikhha, EPFL DATA Lab
- Vojin Jovanovic, EPFL Data Lab & LAMP (Scala) Lab

Template expansion

- One huge step from query plan to C or assembly code.
 - Compile by hand.
- Main advantages:
 - predictable outcome
 - virtually no compilers knowledge required.
- Almost all query compilers are template expanders.
- Two viewpoints: Either
 - 50 years out-of-date technology, or
 - no compiler at all (instead feeds into a compiler).

History of query compilation

- 70ies: Initial prototype of System R: query optimizer => template expansion
- Late 200X: Compilation in stream engines: Streambase, IBM Spade.
- Grust et al., SIGMOD 2009; FERRY
- Ahmad, K., VLDB 2009: DBToaster frontend => triggers => functional code => fusion-enabled compiler => C++ => LLVM.
- Krikellas, Viglas, ICDE 2010: dynamic template expansion.
- Sompolski, Zukowski, Boncz, DaMoN 2011: Compilation in VectorWise.
- Neumann, VLDB 2011: Traditional query optimizer => push engine representation => template expansion to LLVM assembly => LLVM.
 - Some effective fusion due to push-consume pattern.
- 2013: Microsoft Hekaton: compilation in SQL server; Cloudera Impala; MemSQL
- ~2013: Oracle Labs: dynamic compilation: Graal.
 - VM optimization (elimination of unused code paths at runtime).
 - Does not replace a query optimizer / compiler.

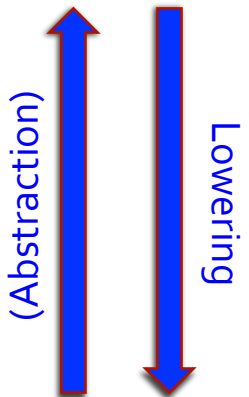
Drawbacks of template expansion

- A conceptually huge step from operator to code template.
 - Very hard to write the templates.
 - A lot of redundancy in the creation of the templates.
 - Brittle, even when compiling to somewhat architecture-independent formats such as C or LLVM assembly.
 - No programming tools support such as strong typing, automatic composition and inlining.
- Difficult to build, extend and maintain.
 - The lesson from System R: compilation was abandoned after initial prototype.

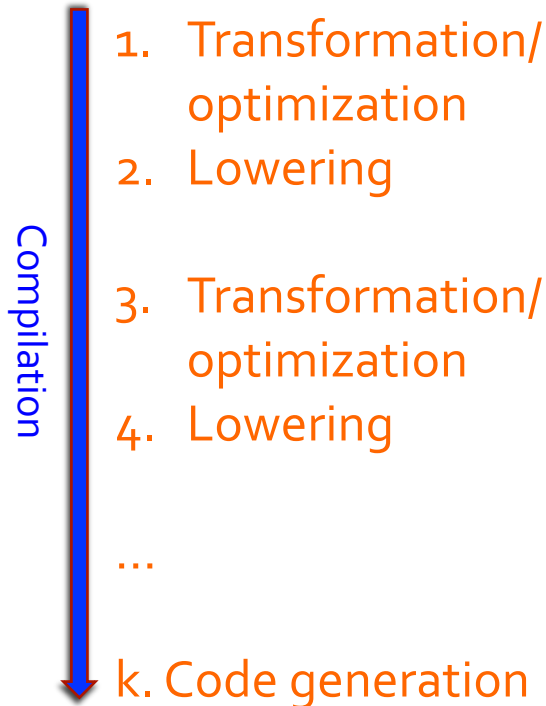
Drawbacks of template expansion

- Operator-by-operator expansion causes us to miss many cross-operator optimization opportunities!
 - vs Fusion in modern compilers: removes the cost of compositionality.
- After template expansion, it is too late for most opts.
 - Code is too large and low-level.
 - As always, our ability to optimize depends on the size of the state space.

High-level vs. low-level code

	code size	optimizations	examples
High-level/ abstract code	Short	Abstraction/Code locality enables high-level optimizations (e.g. join ordering, loop fusion): Small search space.	SQL, collection monad programs (LINQ, Scala Collections, Spark, Cascading, ...)
			
Low-level code	Verbose	Specificity enables low-level optimizations (e.g. register allocation) Search space too large for global optimizations.	LLVM assembly

Stepwise lowering



- Lowering: map to a lower level of abstraction. (lower-level DSL)
 - E.g. by expanding a definition in a library.
- At each level of abstraction, different transformations and optimizations apply.
 - Fusion, inlining, code motion along the way.
- Each lowering level tends to expand the code.
 - Search spaces for optimization expand and only more local and low-level optimizations apply.
- Final level of abstraction is essentially equivalent to the code to be generated.
 - Code generation is simple stringification of final (low-level) AST.

Lowering Examples

Lowering: data structs

```
class Tuple(t: List[Field])
class Relation(schema: List[String],
               tuples: List[Tuple])
val R = new Relation(List("a", "b"),
                     ktuples)
```



```
/* inline: lower to definition */
val R : List[List[Field]] = ...
```



```
/* transform: list to fixed size
array */
var R : Array[Array[Field]] =
  Array.ofDim[Field](k,2); ...
```



```
/* C code generation by
stringification */
Field r[k][2] = { ... };
```

Lowering: loops

```
R.filter(cond: Tuple => Boolean)
```



```
R.flatMap{t => if cond then List(t)
               else List()}
```



```
var l = new List();
R.foreach { t =>
  if (cond(t)) then l.add(t)
}
```



```
var l = new List();
while(R != List()) {
  val t = R.head; R = R.tail;
  if (cond(t)) then l.add(t)
}
```

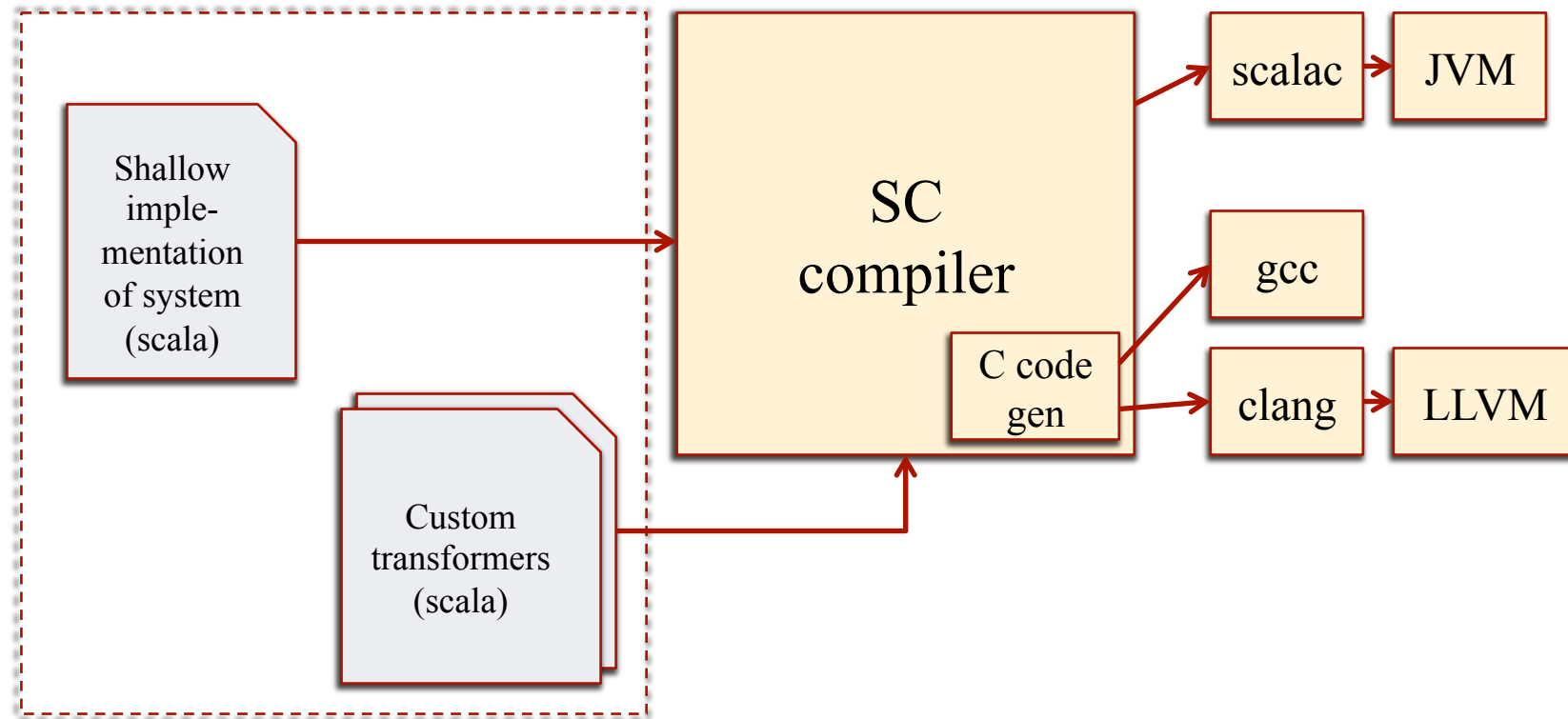
The need for a full PL compiler

- Stepwise lowering requires a powerful intermediate representation.
 - Not relational algebra, not LLVM assembly.
 - Ideally full hybrid functional/imperative PL.
- Different levels of abstraction are DSLs of the same language (DSL = PL + library “vocabulary”)
 - Levels become programmable.
- Inlining: Lift and compile the entire code base.
 - Optimization potential: data structure implementation are the next performance bottleneck in query compilers!
- To get this right, separate compiler infrastructure from query compilation (the application).

The SC DSL compiler

- YinYang [Jovanovic, **CK**, ..., GPCE 2014] & Purgatory:
 - Automatic lifting into IR (“shallow” to “deep”)
 - Finally tagless/polymorphic embedding
 - Easier debugging (simpler types), pattern matching in compiler extensions.
- Programmable/extensible compiler phases.
 - Programmable lowering.
 - Simpler API for compiler transforms.
- Co-designed with the LegoBase system.

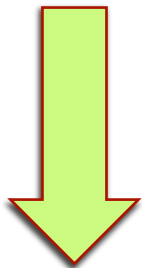
The SC Compiler



Creating DSL compilers with SC

DSL program

(ex.: query)

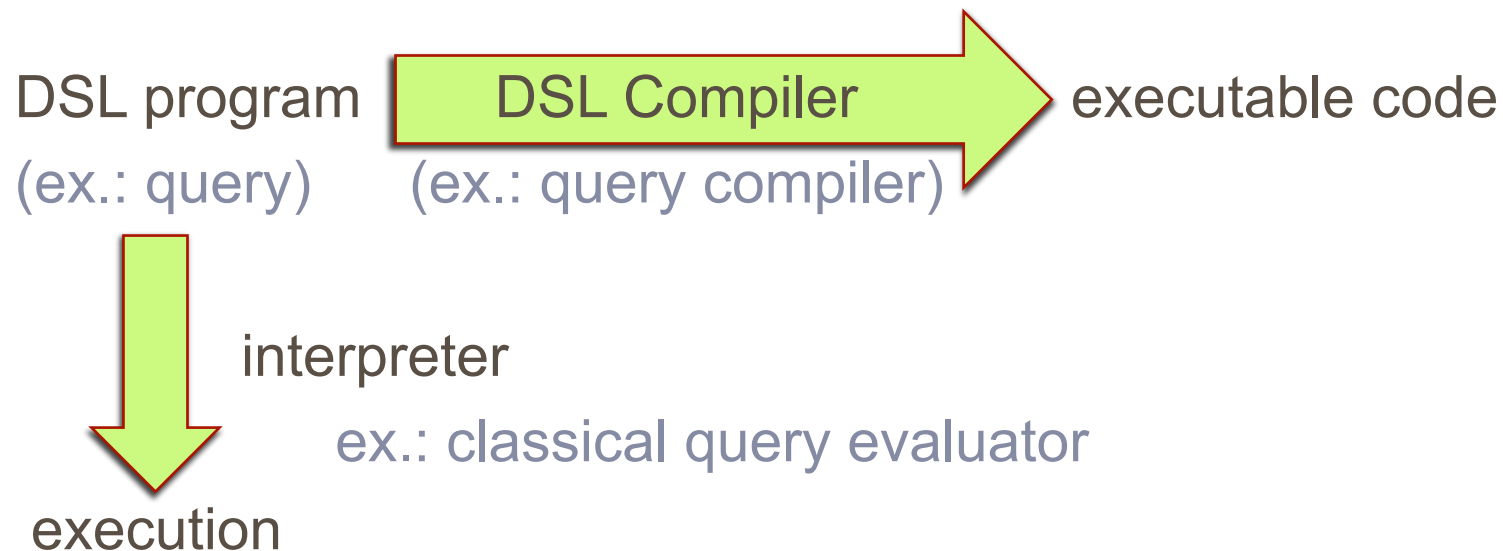


interpreter

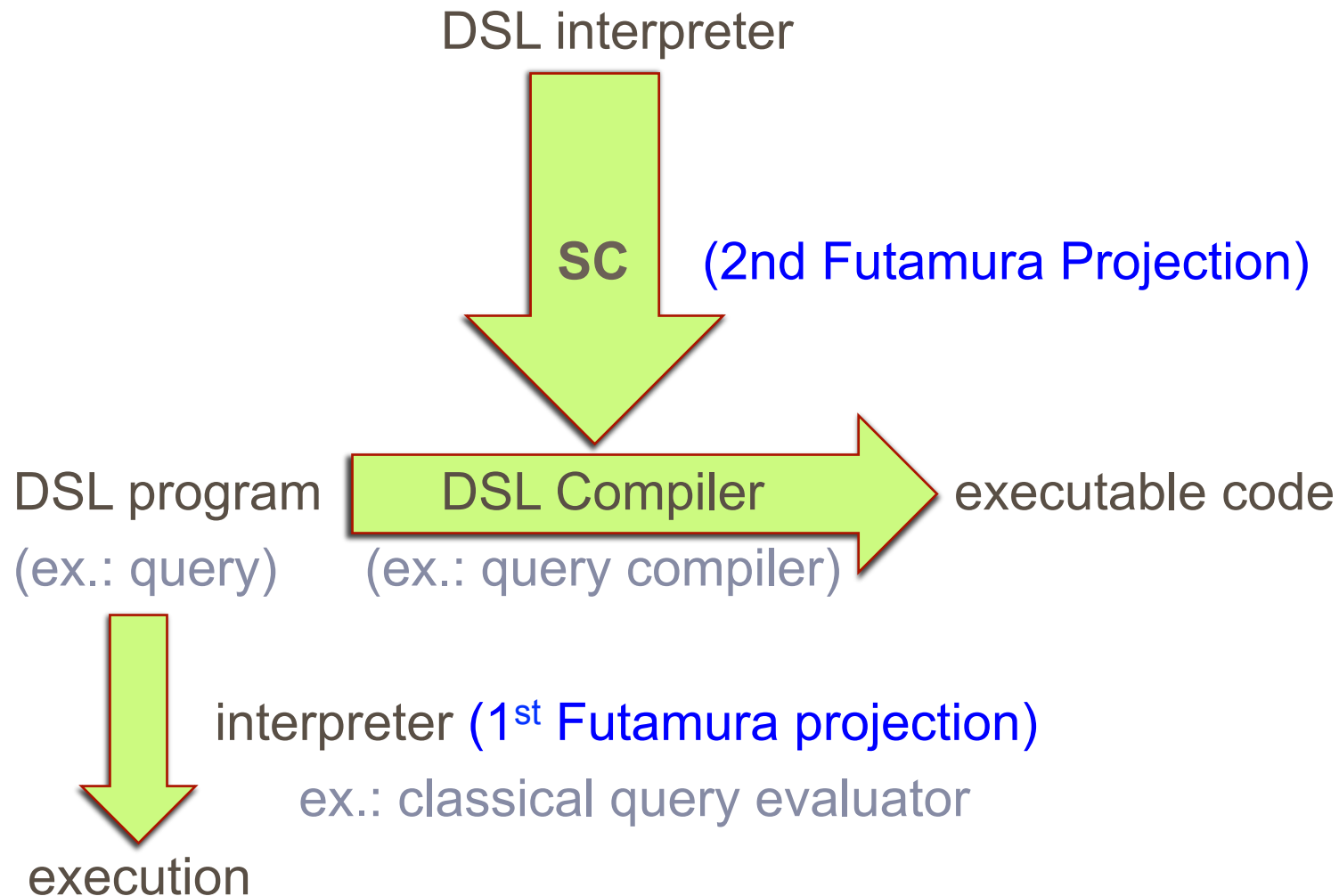
ex.: classical query evaluator

execution

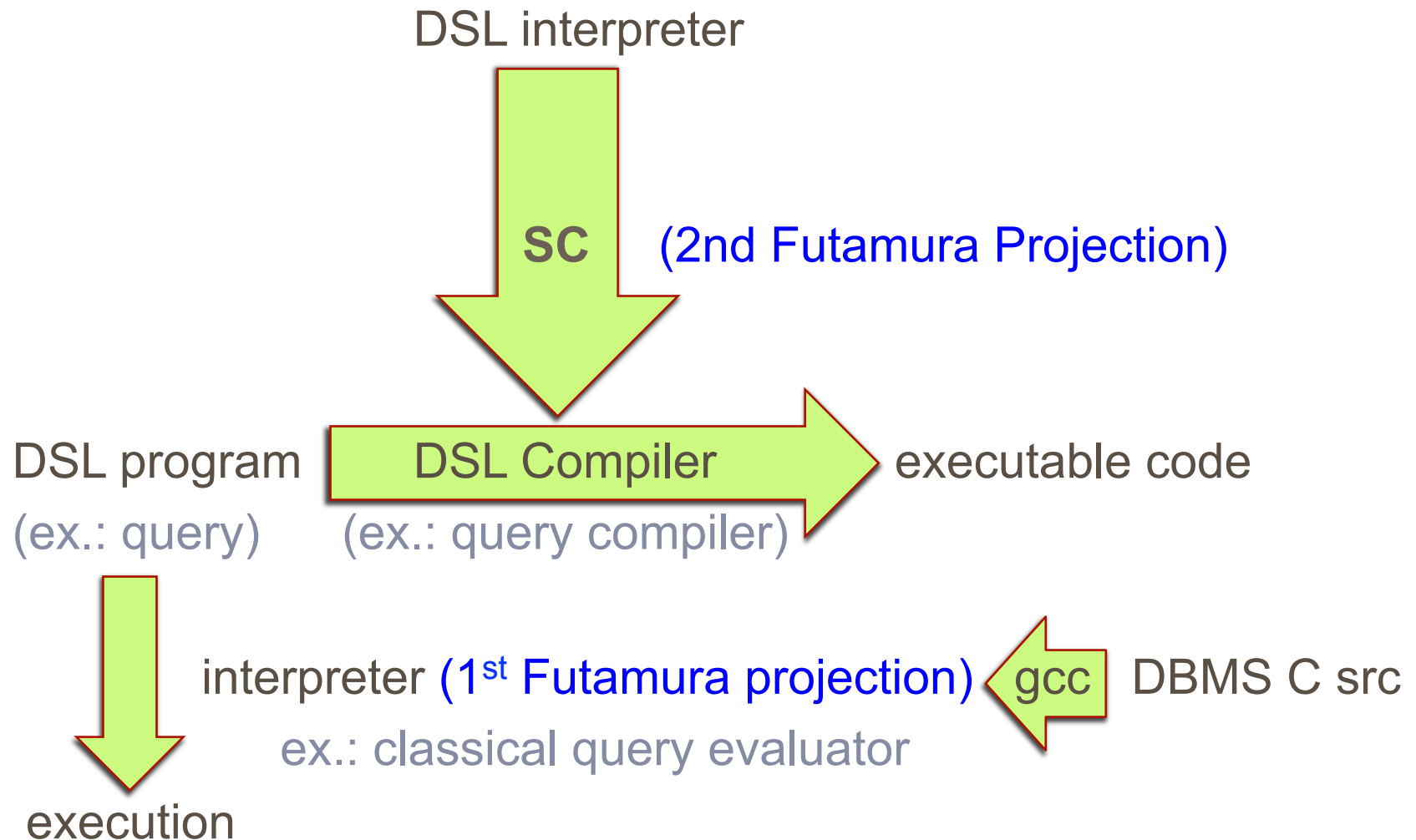
Creating DSL compilers with SC



Creating DSL compilers with SC



Creating DSL compilers with SC



The LegoBase System

Y. Klonatos, T. Rompf, **CK**, H. Chafi, “LegoBase: Building Efficient Query Engines in a High-Level Language”, VLDB 2014.

The LegoBase System

- An ongoing effort to build a database (OLAP) system in a high-level PL (Scala) without performance penalties.
- Use modern DSL compiler framework (SC).
- Produces low-level Scala or C (\Rightarrow gcc/clang \Rightarrow LLVM)
- Current limitations:
 - Single-core main-mem query engine.
 - No classical query optimizer yet. Imports query plans from commercial system.
- Open source (GitHub). See <http://data.epfl.ch/legobase>

```

1 @deep class AggOp[B](child: Operator, grp: Record=>B,
2   aggFuncs: (Record,Double)>=>Double*) extends Operator {
3   val hm = HashMap[B, Array[Double]]()
4
5   def open() { parent.open }
6   def consume(tuple: Record) {
7     val key = grp(tuple)
8     val aggs = hm.getOrElseUpdate(key,
9       new Array[Double](aggFuncs.size))
10    var i: scala.Int = 0
11    aggFuncs.foreach { aggFun =>
12      aggs(i) = aggFun(tuple, aggs(i))
13      i += 1
14    }
15  }
16  def next() : Record = {
17    hm.foreach { pair =>
18      child.consume(new AGGRecord(pair._1, pair._2))
19    }
20  }
21 }

```

Push operator implementation

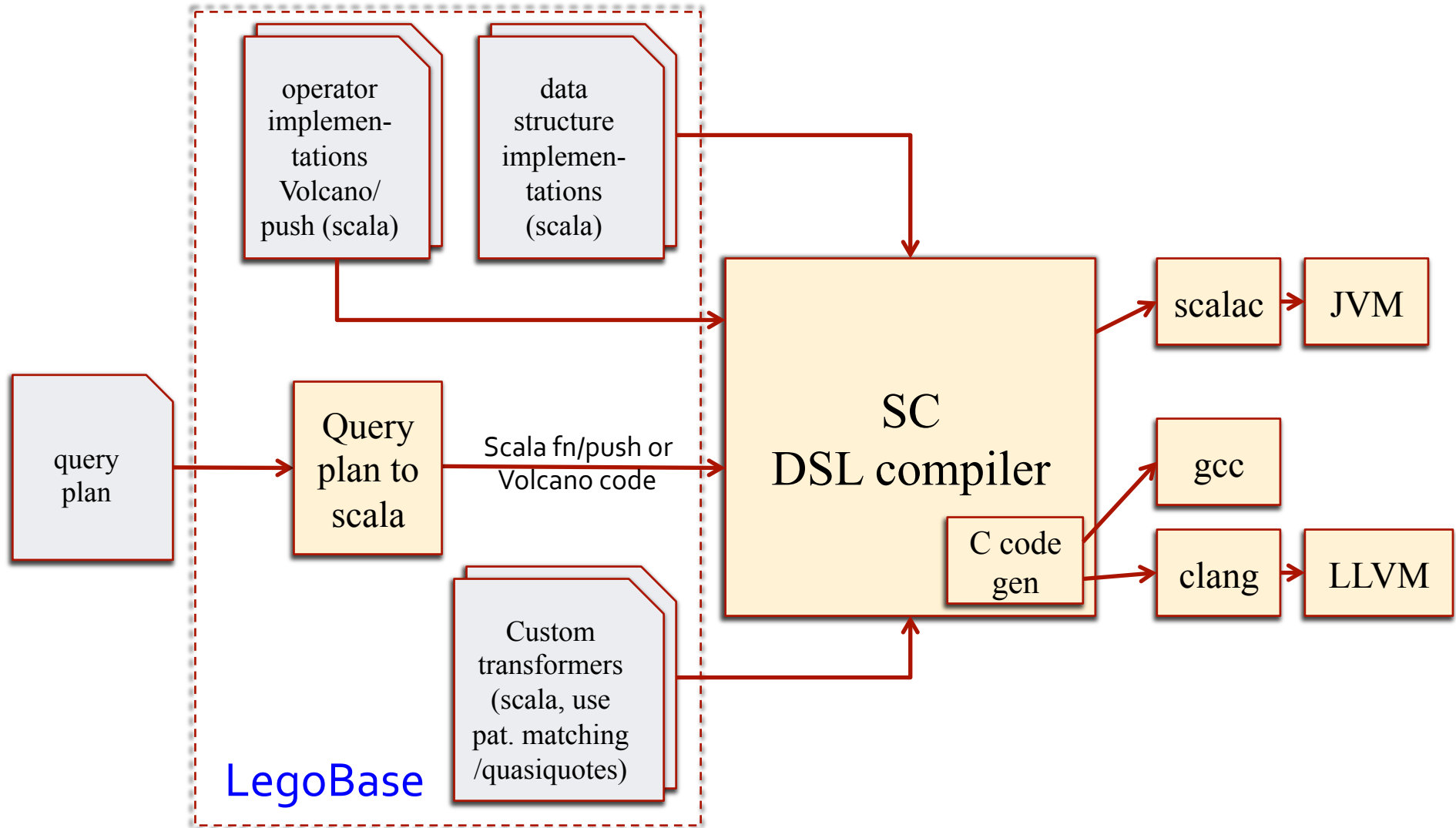
Functional query plan API

```

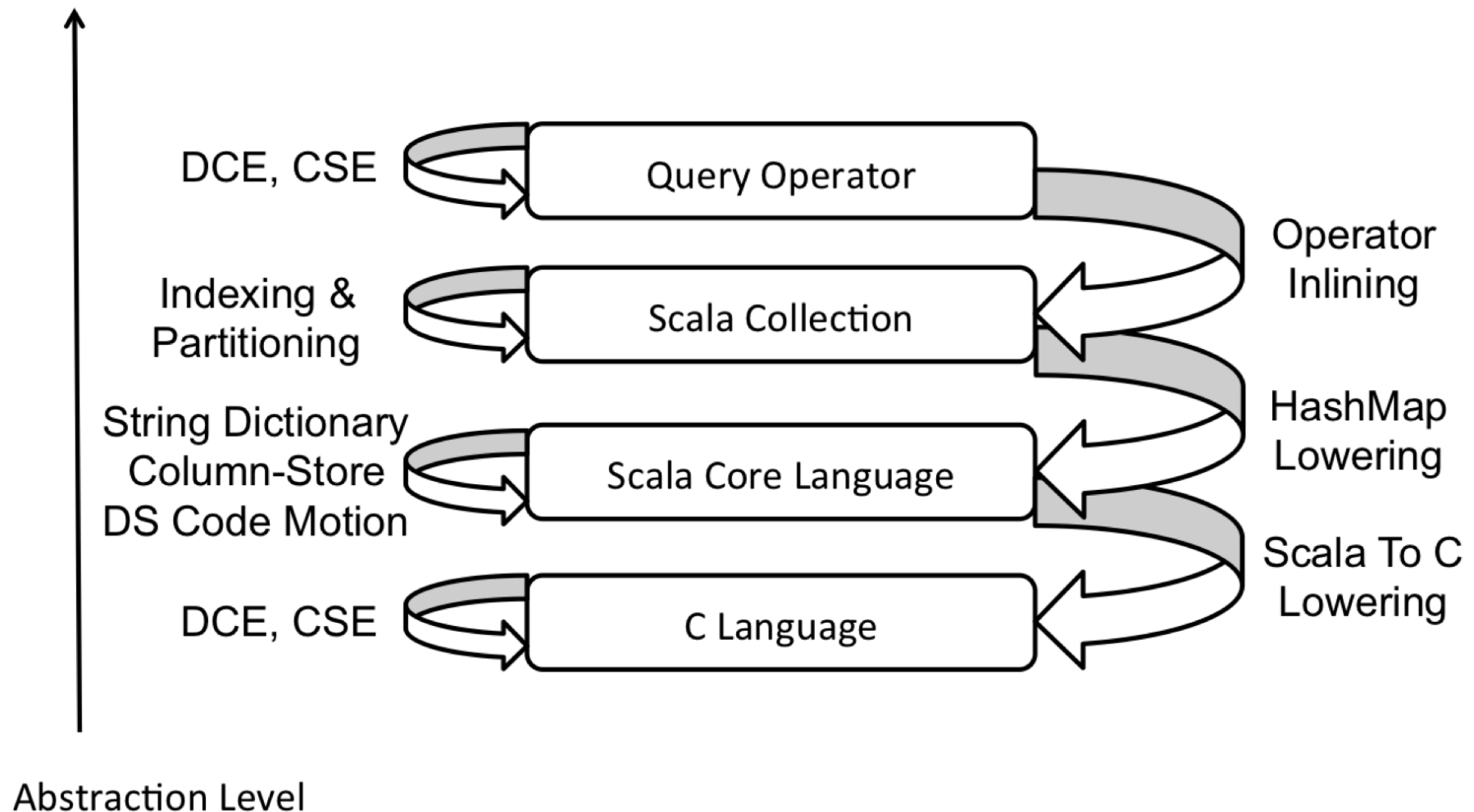
def Q12() {
  val lineitemTable = loadLineitem()
  val ordersTable = loadOrders()
  val ordersScan = new ScanOp(ordersTable)
  val lineitemScan = new ScanOp(lineitemTable)
  val lineitemSelect = new SelectOp(lineitemScan)(x =>
    x.L_RECEIPTDATE>=19940101 && x.L_RECEIPTDATE<19950101 &&
    x.L_SHIPDATE < x.L_COMMITDATE &&
    x.L_COMMITDATE < x.L_RECEIPTDATE &&
    (x.L_SHIPMODE == "MAIL" || x.L_SHIPMODE == "SHIP")
  )
  val jo = new HashJoinOp(ordersScan, lineitemSelect)
    ((x,y) => x.O_ORDERKEY == y.L_ORDERKEY)
    (x => x.O_ORDERKEY)(y => y.L_ORDERKEY)
  val aggOp = new AggOp(jo, 2)(x => x.L_SHIPMODE)(
    (t, agg) => {
      if (t.O_ORDERPRIORITY=="1-URGENT" ||
        t.O_ORDERPRIORITY=="2-HIGH") agg + 1 else agg
    },
    (t, agg) => {
      if (t.O_ORDERPRIORITY!="1-URGENT" &&
        t.O_ORDERPRIORITY!="2-HIGH") agg + 1 else agg
    }
  )
  val sortOp = new SortOp(aggOp)((x, y) => x.key != y.key)
  val po = new PrintOp(sortOp)(kv =>
    printf("%s|%.0f|%.0f\n", kv.key, kv.aggs(0), kv.aggs(1)))
  po.open
  po.next
}

```

The LegoBase System



LegoBase IR Abstraction Levels



LegoBase Transformers

```

pipeline += OperatorInlining

pipeline += SingletonHashMapToValue
pipeline += ConstantSizeArrayToValue
pipeline += ParamPromDCEAndPartiallyEvaluate
if (settings.indexing) {
  pipeline += IndexingAndPartitioning
  pipeline += ParamPromDCEAndPartiallyEvaluate
}

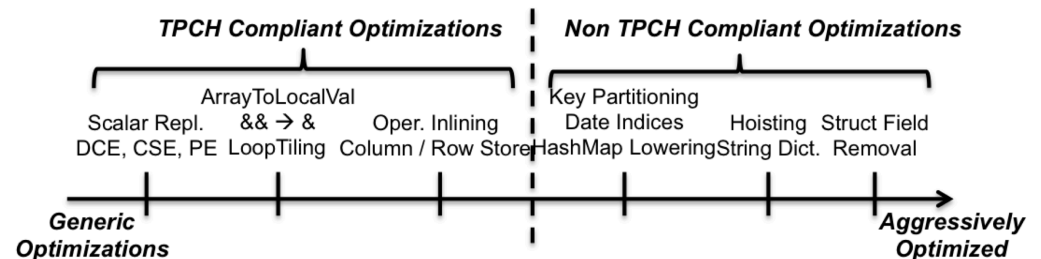
if (settings.hashMapLowering) pipeline += HashMapLowering

if (settings.stringDictionary) pipeline += StringDictionary
if (settings.columnStore) {
  pipeline += ColumnStore
  pipeline += ParamPromDCEAndPartiallyEvaluate
}
if (settings.dsCodeMotion) {
  pipeline += HashMapHoisting
  pipeline += MallocHoisting
  pipeline += ParamPromDCEAndPartiallyEvaluate
}

if (settings.targetIsC) pipeline += ScalaToCLowering

pipeline += ParamPromDCEAndPartiallyEvaluate

```



Performance & productivity

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Scala Library	5918	35855	8746	12445	43469	656	43976	7603	395985	35337	3859
Unoptimized C	812	182	967	2004	943	138	796	735	3621	918	100
TPCH Compliant C	663	166	967	2004	825	55	796	555	2947	756	100
Optimized C	116	8	36	37	39	19	15	78	727	242	17
Handwritten C	71	5	24	61	74	12	45	124	235	129	24

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Scala Library	11426	45912	5652	1382	12993	3260	5665	3513	4853	37207	2277
Unoptimized C	911	3656	367	198	2229	536	1136	656	264	4139	302
TPCH Compliant C	742	3268	198	82	2187	472	1136	583	253	3052	200
Optimized C	60	681	38	62	134	81	305	3	41	96	19
Handwritten C	64	216	64	43	90	28	45	190	102	182	36

LegoBase-shallow #loc (<https://github.com/epfldata/dblab/tree/develop/lego/src/main/scala>)

Operator implementations	384
Storage manager/loader	222
Other (main, ...)	144
Total	750

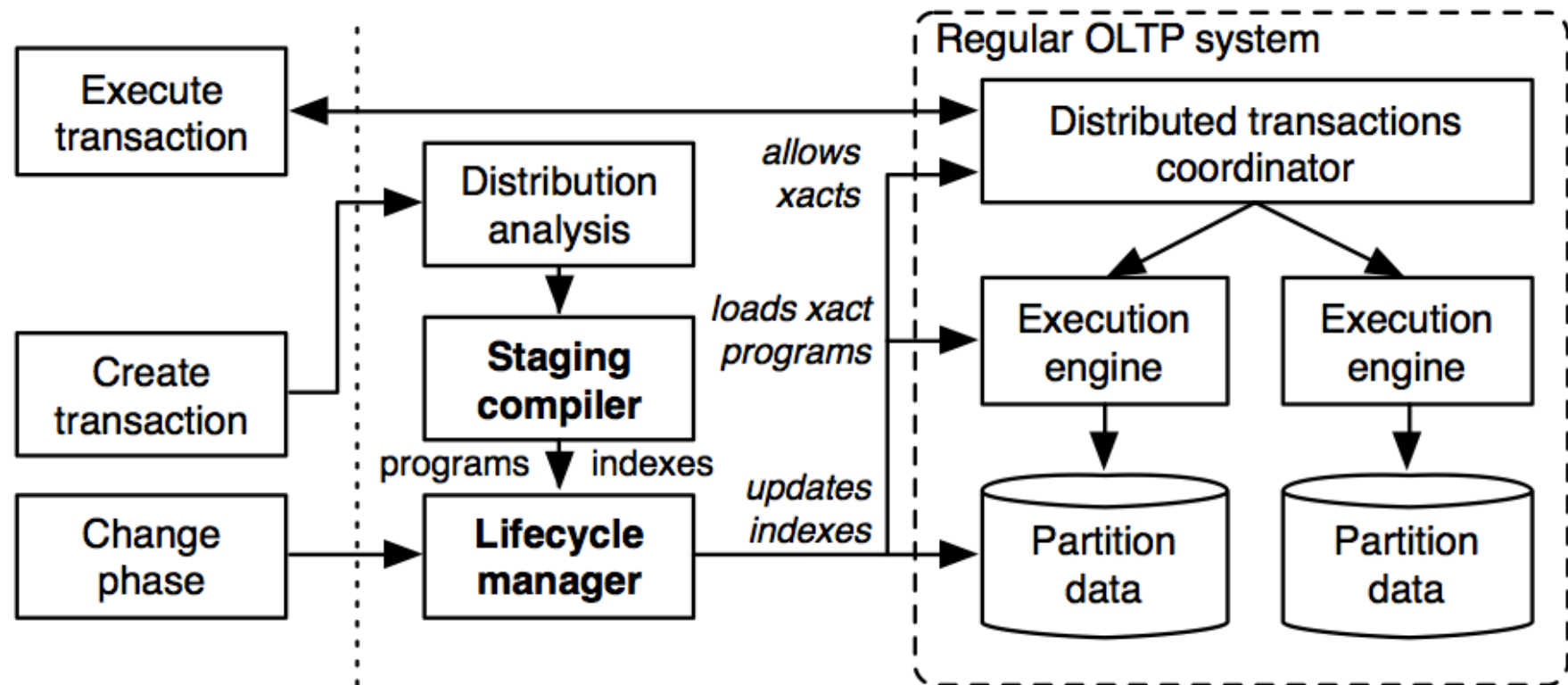
Summary slide - Advantages

- Productivity: built in the high-level PL Scala in a few hundred lines of code.
- Competitive performance
 - Can profit from all the code efficiency benefits of previous query compilers.
 - Leverages additional potential for cross-operator optimization.
 - Queries are inlined into the engine's library code (data structure implementations).
 - Data structure specialization.

The S-Store (OLTP) System

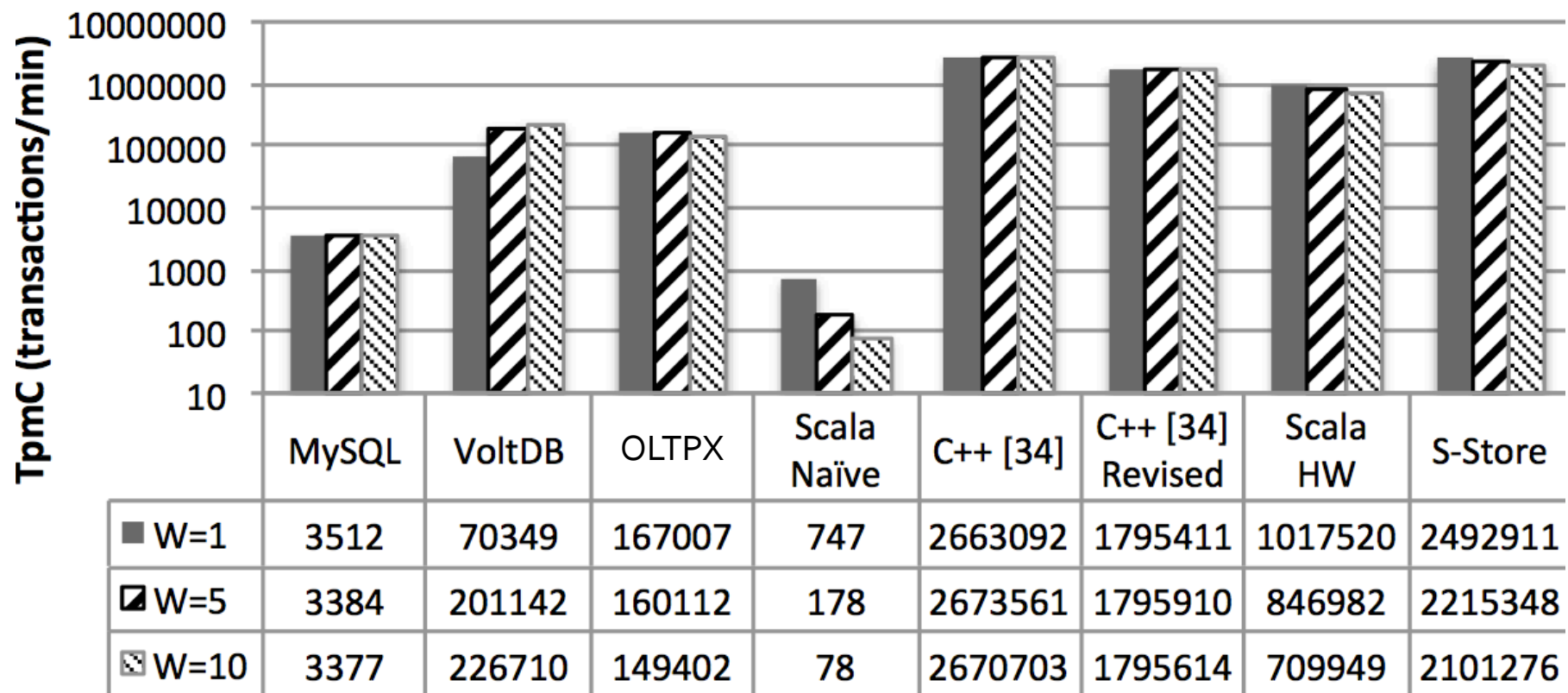
with Mohammad Dashti, Thierry Coppey, EPFL
DATA Lab

Architecture of the S-Store System



Inspired by H-Store. Currently only 1-core serial execution

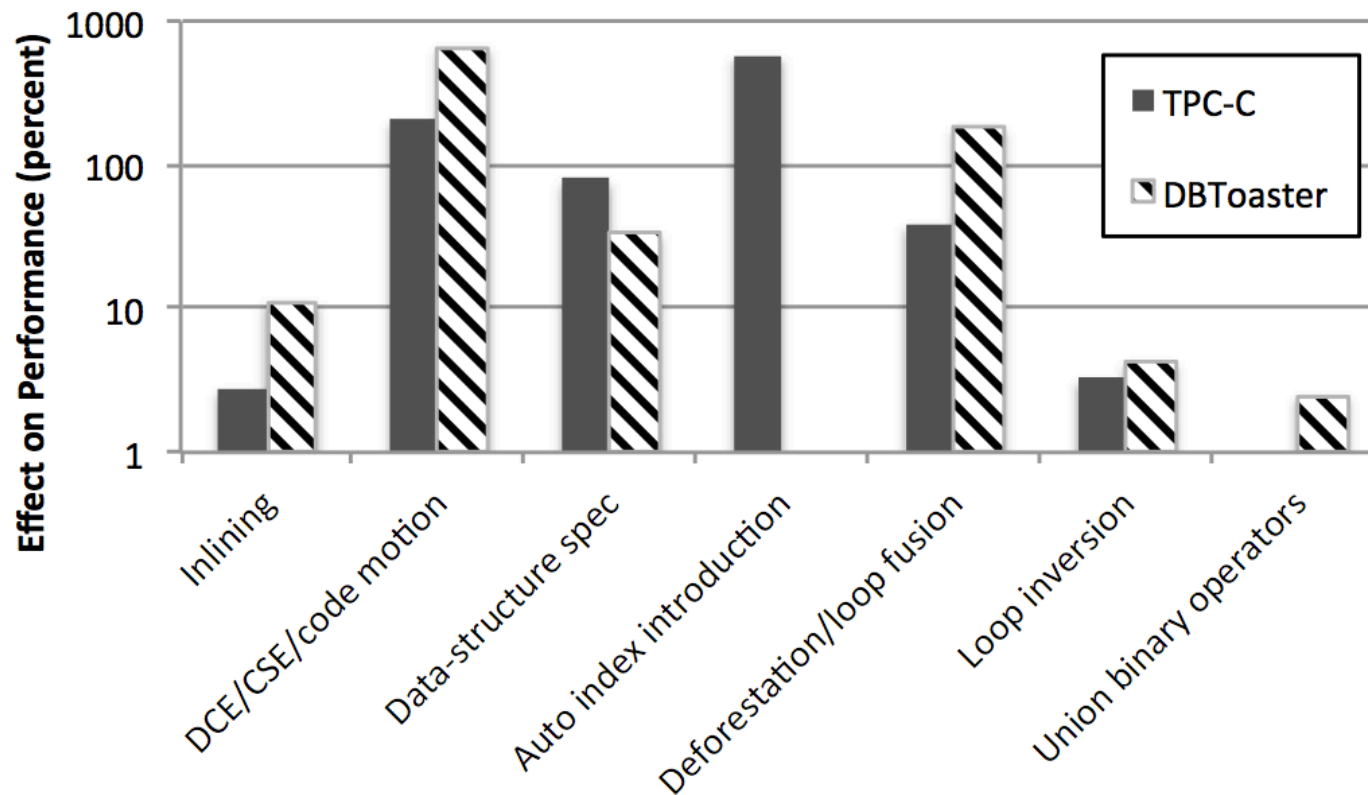
TPC-C benchmark results



TPC-C: opts in handwritten impl and in S-Store

TPC-C transaction	Version	Mutable record manipulation	Inlining	Common subexpr. elim.	Data structure specialization	Automatic index introduction	Deforestation	Loop inversion
NewOrder	S-Store	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	—	✓	✗	✓	✗
Payment	S-Store	✓	✓	✓	✓	✓	✓	✗
	Hand-written	✓	✗	—	✓	✗	✓	✗
OrderStatus	S-Store	✗	✓	✓	✓	✓	✓	✗
	Hand-written	✗	✗	—	✓	✗	✗	✗
Delivery	S-Store	✓	✓	✓	✓	✓	✓	✓
	Hand-written	✓	✗	—	✓	✗	✓	✗
StockLevel	S-Store	✗	✓	✓	✓	✓	✓	✓
	Hand-written	✗	✗	—	✓	✗	✓	✗

Impact of S-Store by optimization



Conclusions

- Abstraction without regret:
 - Can expert programmer skill be automatized? We think so.
 - Great benefit in programmer productivity.
- This is not just about programmer productivity.
 - Human experts do have good and bad days.
 - The compiler is more disciplined than a human.
- Future DBMS that do NOT use compiler technology extensively will not be performance competitive.

Vision

- PL and Compilers will change the way we built DBMS.
 - We will implement DBMS in high-level languages.
- Compiler-system co-design
 - From complex monolithic systems to **libraries** for operators, concurrency primitives, optimizers; compiler plugins (code transformers).
- Frameworks, libs, and toolkits for DBMS construction.
 - Accelerate research
 - Towards better experimental reproducibility
 - Agile DBMS construction&specializ'n (*the DBMS factory is the product*).
 - <http://github.com/epfldata/dblab/>

Thank you / questions

Main publications related to the talk

- **CK**: “Abstraction Without Regret in Database Systems Building: a Manifesto”. IEEE Data Eng. Bull. 37(1): 70-79 (2014).
- Y. Klonatos, T. Rompf, **CK**, H. Chafi, “LegoBase: Building Efficient Query Engines in a High-Level Language”, VLDB 2014.
- M. Dashti, T. Coppey, V. Jovanovich, **CK**: “Compiling Transaction Programs”. Manuscript, 2014.
- T. Rompf, N. Amin, T. Coppey, M. Dashti, M. Jonnalagedda, Y. Klonatos, M. Odersky, **CK**. “Abstraction Without Regret for Efficient Data Processing”. DCP 2014.
- V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, **CK**, M. Odersky: “Yin-yang: concealing the deep embedding of DSLs”. GPCE 2014.
- **CK**, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, A. Shaikhha: “DBToaster: higher-order delta processing for dynamic, frequently fresh views”. VLDB J. 23(2): 253-278 (2014).
- **CK**: “Abstraction without regret in data management systems”. CIDR 2013.
- **CK**: “Incremental query evaluation in a ring of databases”. PODS 2010: 87-98.