Abstraction without regret in data management systems

Christoph Koch, EPFL DATA Lab
christoph.koch@epfl.ch

1. MOTIVATING ABSTRACTION

The long-term impact of any research community depends on the timelessness and power of the ideas, concepts, and techniques developed by it. Research that fails to be sufficiently original and that does not create deep insight is likely not to have – and arguably does not deserve – substantial long-term impact.

There are timeless classics among systems papers that stand as examples of what systems research can be and that have motivated countless young researchers to have a career in this field.

But there seems to be a problem in data management research today, best evidenced by the following observation. At some point, most researchers and all academics get into the situation to be asked to recommend a representative set of classic database papers to researchers in other fields or to create a reading list of classics for an advanced course. In that case, we frequently end up with a set of quite old papers (c.f. e.g. the Red Book [8]). These picks have a tendency to be co-authored by Jim Gray. If the reading list is for a broader systems course, we end up struggling to come up with a set of papers that can stand up against the best papers from SOSP and SIGCOMM, conferences that seem to publish fundamental, clean papers more frequently, even in recent years.[1] (Researchers in these communities may see things differently [12].) Most papers do not qualify for such a selection as they cater to short term fashions or simply do not aim hard enough at providing definitive solutions.

We sometimes think of our community as one that is not very good at citing properly and giving credit. In fact, citing fairly is hard because it is frequently difficult to judge a paper's merits and novelty as ideas are not sufficiently worked out and their deep, rather than superficial, relationship to previous work is not presented in the paper. What is frequently lacking is a conclusive deconstruction of the research contribution into first principles and fundamental patterns from which the contribution is composed.

We can hardly blame individual researchers since we, as a community, have made little effort to extract the fundamental principles underlying our work. We can even see this in our courses and textbooks. While several of our textbooks bear *Principles* and *Concepts* in their names, they contain some fundamental concepts (such as transactions), but also plenty of recipes, which on further analysis easily decompose into more fundamental ideas. I will give an example – out-of-core algorithms – to illustrate this in the next section.

For comparison, the systems community is ahead of us here. Systems courses are frequently structured to map course modules to fundamental principles such as aiming for abstraction (modularity, layering), virtualization, client-server, the end-to-end principle, and so on. Frequently, such courses manage to do completely without recipes lectures. Textbooks like [17] aim to address systems from such a pure standpoint.

Just like the publication of Gamma et al.'s *Design Patterns* book [5] has had great impact in software engineering, we should look for data management's own fundamental patterns and principles. In many cases, they may be simple compared to the research in which they are currently buried. I believe this is a necessary exercise, since our community has not focused enough on identifying its principles in their full generality so far.

A research community which does not succeed in making a claim on fundamentals and in exposing its results to the rest of the world in sufficient clarity risks becoming obsolete, with its best work being reinvented elsewhere without the original authors getting credit. Some of this happened recently as part of the big data revolution (cf. [4]). Aiming to discover a core of data management beyond our previous efforts is the best way of protecting the same efforts.

2. EXAMPLE: LOCALITY PRINCIPLES

The ability to turn insights regarding the cost of *moving data* (inside the memory hierarchy and across machines) into high-performance systems and algorithms is a key competence of our research community. Nevertheless, even our textbooks provide little more than recipes for solving some concrete technical problems, such as sorting on a hard drive. But what are the fundamental principles, design patterns, and methods underlying the construction of efficient systems and algorithms that are aware of the memory hierarchy and storage devices? Currently, when researchers face the challenge of designing a, say, join or sorting algorithm for a new kind of device, they are completely on their own; there is not even a methodology to address the problem. I believe that this is a great field for fundamental research.

The systems research community is somewhat ahead of us in that it identifies exploiting locality as fundamental to performance [17] but equates it with the caching principle

[1]This is a lesson from the trenches as I currently co-teach a Principles of Systems course with several systems professors and have struggled to create such a list. I challenge the reader to come up with a representative list of papers isolating principles of data management systems in the spirit of systems papers such as [9, 16].

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

[3]. It appears that there are other principles underlying data locality than caching; for example, the GRACE hash join employs partitioning to make more focussed use of main memory. Many parallel systems employ partitioning to move the computation closer to the data and perform divide-and-conquer in parallel. Join and sorting algorithms on hard drives optimize for sequential rather than random access. An argument for optimizing access to collocated data can be made for other forms of storage such as flash as well, and even main memory (because of block transfers and translation lookaside buffers). None of these ideas are subsumed by caching (although TLBs themselves are caches).

On a high level, we can easily identify three principles: (1) caching, (2) partitioning, and (3) respecting co-locality of storage in data transfers between levels of the memory hierarchy (one implementation of the latter is sequential access to hard drives). There are probably more, and some of the above (partitioning?) should not be treated as just one.

We can turn principles into basic design patterns. For example, a pattern that combines the first and the third principle is: turning a scan of a collection of data on one level of the memory hierarchy into a hierarchical scan that employs a block of memory on the next higher level of the memory hierarchy (smaller but faster) as a buffer, transferring collocated data at the granularity of blocks.

The key out-of-core algorithms such as a block-nested loop join or a GRACE hash join, which we consider fundamental contributions of our community, are compositions of more fundamental ideas. It is striking that our textbooks report such recipes, but do not attempt to deconstruct them into the fundamental constituents that they are composed of.

For instance, the block-nested loops (BNL) join can be obtained as the result of the composition of the naive nested loops join

```
for each tuple r of relation R {
  seek r; transfer r;
  for each tuple s of relation S {
    seek s; transfer s;
      if cond(r, s) then output (r,s)      }}
```

with the hierarchical scan pattern (twice), which results in

```
for each block BR of relation R {
  seek begin of BR; transfer BR;
  for each tuple r of BR {
    for each block BS of relation S {
      seek begin of BS; transfer BS;
      for each tuple s of BS {
        if cond(r, s) then output (r,s)    }}}}
```

and then moving the second for-loop (looping over individual tuples r of block BR) in.

The reader can verify the author's intuition that most if not all of the proposed join algorithms in the literature are, assuming suitable formalism, compositions of much simpler algorithms on a random-access memory model with quite orthogonal ideas that relate closely to the storage device or memory hierarchy. The latter are often the same for other out-of-core algorithms on the same storage device/memory hierarchy. Thus there are in the product more out-of-core algorithms than there are basic algorithmic ideas and device models. These could be put into a library that is potentially much smaller and simpler than the literature on the topic! Also, as a new device (such as, recently, flash) comes along, we have a methodical and productive way of getting good out-of-core algorithms for the device: by composing the (memory-hierarchy oblivious) basic algorithm with specific patterns that capture the peculiarities of the device. It should be possible to perform this form of composition automatically.

3. COMPOSING SYSTEMS

The goal of this paper is not to give a polemic about research in our community but to discuss a vision of performing research and building systems more abstractly, without regret (= a performance penalty).

I argue that much of our past research work has effectively – sometimes unknowingly – performed manual composition of more fundamental ideas and patterns.

At a recent major database conference, a keynote on advice for young researchers suggested that there are *number of data models* times *number of technical challenges in databases* many database papers to be written, and whenever a new data model arrives, we can write plenty more papers. I believe that many of us tap into this source of paper topics. This lends support to my claim that much published research is the composition of more fundamental ideas. Taking the argument in reverse, there should be a relatively small core of fundamentals to much of our work.

These fundamental building blocks should be identified, consolidated, and collected for two reasons. First, along the lines discussed above, the consolidation may guide our future research and steer our activities towards creative original work – it will be much simpler to tell which work is incremental. It will allow us to better understand our contributions in their full generality, and how they interact in generality with other fundamental patterns. This is ultimately key to allowing for our work to remain relevant in the face of change.

But in addition to this, if we *put these core building blocks into a software library*, we can resort to it to perform such composition automatically in the future. The desired composition can be achieved by implementing a system in a high-level programming language, using this library. The composition is specified using constructs of modern object-oriented programming languages such as association, aggregation, genericity, and composition through multiple subtyping (of e.g. Java interfaces or mix-in composition using Scala traits) and various more advanced patterns of composition that can be defined in such languages [5]. The composition is then performed by a compiler.

This clashes with the received wisdom in systems construction, which dictates that while modern compilers have freed us from the need to write assembly code in most scenarios, they are not good enough to allow for truly high-level programming without incurring a substantial performance penalty. Systems programming, with a few exceptions, has remained the domain of the C language.

Programmers continue to perform manual code inlining on a large scale, with all the productivity issues this entails. For instance, in Postgres, there are tens of slightly divergent copies of the code corresponding to the abstractions of pages and B-trees, as a way of achieving performance, not as a certificate of disorganized growth. The main commercial RDBMS systems consist of millions of lines of code each, which suggests that similar inlining is happening there.

The received wisdom is also that the concurrency control/buffer management/storage and indexing subsystem of a DBMS is notoriously monolithic, heavily interdependent, and thus a great challenge to build and get right (cf. [10]). Of course we could build the three components generically and individually, with clean interfaces, and then employ an optimizing compiler to aggressively inline them. It would be worth understanding whether automatic composition by modern compilers is still not good enough to follow such a path today, and what would need to be improved.

There are many cases where automatic composition of clean, simple, and generic components by an optimizing compiler may lead to systems that are just as fast or better than those where this composition is done by humans; in addition, there is a very substantial argument for letting the compiler do the work; we may cite productivity and our wish to do work that is future-proof (the composition can be automatically done by the compiler again in a changed environment where the mix of input components is different).

A classical success story in this space is Singularity [7], an operating system entirely written in a high-level language and run as managed code (in a virtual machine). Singularity is more efficient at many key OS tasks than any of the mainstream OS, and does not rely on the compiler to perform magic: instead, it exploits insights obtained by static analysis to avoid doing some work at runtime. Clearly, it would be worth attempting to build database systems along these lines.

In the remainder of this paper, I will take another path. I will argue that modern extensible, staged compilers informed by our intervention will allow to perform automatic composition without performance penalty, achieving *abstraction without regret*. This concerns contributions of relatively small (such as join algorithms) to very large complexity (such as complete database systems).

A successful example of this already exists in the form of SPIRAL, a synthesis tool for generating extremely efficient code (at least as good as the code human experts can realistically write) from high-level programs in the domain of signal processing [13]. It is based on extending compilers by domain-specific optimization ideas. Somewhat similar efforts have been made in the context of query optimization [6], though with arguably more moderate success. The reason why I think the time is ripe to reconsider this path is that now we are getting help from the PL and compilers communities. Efforts are underway [15, 2] to build extensible compilers of complete high-level programming languages as libraries that large communities of developers will lead to maturity, lowering the challenge for us to make composition fast in our domain.

4. LMS: THE COMPILER AS A LIBRARY

With the latest generation of programming languages, language virtualization and powerful features for software composition have come of age. These features allow us to offer optimizing compilers *as extensible libraries*. EPFL's LMS [15] and Stanford's Delite [2] are examples of such libraries. By their extensibility, they add a further quality to the offerings of optimizing (just-in-time) compilers such as those of LLVM and Java Hotspot. By their generality – supporting all of a general-purpose language (Scala) – they can be used to compile and optimize both entire data management systems and queries processed by those in one uniform framework. For comparison, extensible optimizers such as Starburst [6] can only handle the latter. The attractive promise of extensible compilers is that they may eliminate all the adverse effects of programming in the abstract by automatic composition, aggressive inlining, loop unrolling, and fusion; this has been referred to as *abstraction without regret.*

Such compilers-as-libraries are an interesting resource for constructing and composing (data management) systems. With relatively little effort, we are able to build systems on a higher level and in a radically simpler way than in the past, without neglecting performance.

Garbage-collected memory management, functional programming, and object orientation in modern programming languages create many potential inefficiencies such as unnecessary object creation, data (structure) copying, and, in consequence, suboptimal data locality. Compilers need to eliminate as many of these as possible for performance. Recently, just-in-time compilation and adaptive optimization in the virtual machine have been employed to push the state of the art, for example in LLVM and Java Hotspot.

Lightweight Modular Staging (LMS) is a library built in Scala that performs many of the typical optimizations implemented in these systems on an abstract syntax tree (AST) of Scala. This AST is part of the LMS library, and, using language virtualization, LMS is also able to lift Scala code to an AST representation. That is, it does not require, or implement, a separate Scala parser: A Scala program compiled with LMS will execute with a redefined "semantics of Scala" that, rather than evaluating the Scala code, turns it into an AST. Of course, a Scala program can employ a controlled mix of Scala code that is processed in the normal way and code that will be turned into an AST for processing by LMS.

The optimizations currently implemented in LMS include duplicate expression elimination, loop fusion, also for higher-order functions such as map and filter on collection types, loop and recursion unfolding, and turning code into administrative normal form, which makes it easy to generate imperative code from functional programs. (Similarly, LLVM creates static single assignment form code representations).

Compared to LLVM and Hotspot, the advantage of LMS is its easy extensibility – any optimization can be overridden or refined using mix-in composition. Moreover, the result of a set of optimizations is a Scala AST, which can be interpreted, for which code can be generated in Scala or other languages (such as C), or which can be further optimized in another stage of LMS. This allows to build static and just-in-time compilers with various flavors of adaptivity.

The extensibility of LMS in a sense guarantees that abstraction without regret is achievable: if we find that the optimizing compiler does not produce good enough code in a scenario, the library allows us to override or refine its optimizations at an arbitrarily fine-grained level. Not just can we replace optimizations at the level of types, but we can catch and override individual (conditional) cases of pattern matching of the AST.

5. STAGED COMPILATION OF SYSTEMS

Staged compilation is not new to databases: Since the earliest times of System R, database queries have been just-in-time (JIT) compiled, at query time, from the SQL virtual machine to executable machine code. A number of recent papers have returned to this idea [14, 1, 11]. Using LMS, we can perform staged compilation of the entire system's code, rather then just the queries, which allows us to do substantially more. Specifically, we can write the entire system in high-level code, and rely on the compilation framework to create code at the level of efficiency achieved by en experienced systems programmer using a low-level language.

Meta-levels, and bridging them. In query engines, we usually process queries in multiple stages, each with their own distinct internal representations. An RDBMS usually has distinct intermediate representations (IRs) for SQL, augmented SQL, the query algebra, and query plans. The above query JIT compilers have additional stages for low-level code representations.

LMS admits a subtle but fundamental deviation from this:

The IRs of the various stages are all instances of the Scala AST, with different IR-specific data types. For example, in a relational algebra stage, we are not dealing with a separate relational algebra language but with Scala objects from a relational algebra operator class hierarchy.

Traditionally, we do not take such an approach because it would require us to write an optimizer for a much larger language (than we traditionally feel necessary). With LMS, we are provided with a complete optimizing compilation framework for Scala, so this is not a worry.

This has a number of interesting consequences. Traditionally, we are forced to a certain sequence of compiler stages because we switch IRs between them. Now, using the same IR throughout compilation, we are free to choose stages either for modularity – for handling code complexity – or to defer parts of compilation to a later time (e.g. for just-in-time compilation).

More fundamentally, this allows us to deal with multiple meta-levels in the same compiler and even to bridge these levels: Observe that, above, we have been quite carelessly mixing two meta-levels in our discussion of compilation. We called for the application of optimizing compilation both to queries – as work such as [14, 1, 11] does – and to the database systems themselves, that is, both to a classical database system's source code and to the query plans that they maintain and interpret, at runtime, in data structures. Our approach here allows us to do more than just (independently) compile at several meta-levels at once: It allows us to bridge meta-levels, allowing us to do unusual and interesting things within the compilation framework. Two examples:

Breaking the meta-wall 1: code generation and algorithm synthesis. Consider the kind of synthesis of out-of-core algorithms that was described in Section 2 – from simple abstract algorithms, descriptions of the machine/memory hierarchy, and simple memory-hierarchy related optimization patterns. If we consider for instance an index-nested loops join, the choice to use an index results from optimization, and is part of the query plan meta-level; at the same time, the join operator code lives in the query engine. Keeping a strict separation between meta-levels (and not staging the compilation of all of the code involved), the kind of synthesis we have suggested would not be possible.

For comparison, a classical query compiler would transform and optimize a query plan; in the final stage, it would perform code generation for the optimized plan. The resulting code lives on the lower meta-level, but is not available for further optimization within the framework.

Breaking the meta-wall 2: synthesizing cost functions. The availability of the AST of the DBMS's entire codebase, if we want it, also allows us to implement interesting transformers and interpreters with unusual semantics. For example, we can write – and should make part of the library – a form of interpreter for cost function synthesis: This one simply traverses the AST and coarsens it to a cost function. For an expression e of the form "loop 500 times { e' }", we can coarsen to $cost(e) = 500 * cost(e')$; the cost of certain nodes in the AST will be zero, while others, such as a system call accessing a disk, will weigh in. Costing recursive functions will involve solving recurrences, which in the case of most DBMS primarily applies to traversing B-trees and plan trees (and here the recurrences are simple). Cost function synthesis involves two meta-levels: part of the code that we coarsen to a cost function is from the database system, but the cost function is for (part of) a query plan, and can be used in a query engine that interprets that plan.

6. SUMMARY

This paper has argued for us to work at a greater level of abstraction, both in our research and in the systems that we build. There is a need to revisit data management research with an aim to identify fundamental principles and patterns. I have illustrated that this is a low hanging fruit even in the space of exploiting data locality, which appears well explored and central to all performance considerations in data management. I have argued that understanding the fundamentals of our area is central to protecting our past investment and to doing better research in the future. It will also increase our productivity in systems development without necessarily causing a performance penalty. To this end, I have motivated a number of research directions, from the automatic synthesis of out-of-core algorithms over work in the spirit of the Singularity operating system to using extensible compiler frameworks such as LMS for systems development.

Acknowledgments

I thank Yannis Klonatos, Viktor Kuncak, Andres Nötzli, and Andrej Spielmann for insightful discussions.

7. REFERENCES

[1] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. In *Proc. VLDB*, 2012.

[2] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, pages 89–100, 2011.

[3] P. J. Denning. The locality principle. *C.ACM*, 2005.

[4] D. DeWitt and M. Stonebraker. Mapreduce – a major step backward. In *Database Column (Blog)*, 2008.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *SIGMOD Conference*, pages 377–388, 1989.

[7] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41:37–49, 2007.

[8] M. S. Joseph M. Hellerstein. *Readings in Database Systems, 4th edition*. MIT Press, 2005.

[9] B. W. Lampson. Hints for computer systems design. In *Proc. SOSP*, 1983.

[10] D. B. Lomet, K. Tzoumas, and M. J. Zwilling. Implementing performance competitive logical recovery. *PVLDB*, 4(7):430–439, 2011.

[11] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[12] R. Pike. Systems software research is irrelevant, Feb. 2000. http://herpolhode.com/rob/utah2000.pdf.

[13] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of Parallel Computing*, chapter Spiral. Springer, 2011.

[14] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled query execution engine using JVM. In *Proc. ICDE*, 2006.

[15] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[16] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. In *Proc. ICDCS*, 1981.

[17] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, 2009.