Collaborative Data Science using Scalable Homoiconicity

Holger Pirk
Imperial College London
hlgr@imperial.ac.uk

**Motivation:** Data science is increasingly collaborative. On the one hand, results need to be distributed, e.g., as interactive visualizations. On the other, collaboration in the data development process improves quality and timeliness. This can take many forms: partitioning a problem and working on aspects in parallel, exploring different solutions or reviewing someone else’s work.

While the benefits of tool support for collaborative software development are established, the existing tools do not meet the demands of data scientists. They struggle to manage complex data processing pipelines without a clear notion of correctness on large and dirty datasets. While classic data management systems have limited support for “collaboration” in the form of concurrent clients, they present the same view of the data to all users (most of the time). Data science has different requirements: first, users may operate “offline,” i.e., on a local copy of the dataset. Second, even when online, developing a data processing pipeline on a dataset concurrently modified by others hurts productivity.

**State of the Art** projects like OrpheusDB [13], Dolt [2], DVC [1] and Pachyderm [9] have recognized the need for versioning in data science and provide the infrastructure to store multiple versions of a dataset. Where they fall short is the handling of branching versions, in particular resolving conflicts during merges. To be practical, a collaborative data science system must make trivial merges fast, automate as many merges as possible and provide tool support when merges need manual intervention. Existing systems do not fulfill these requirements: they have no notion of concurrent versions, present diffs at tuple-granularity and require mostly manual conflict-resolution. Sophisticated merge strategies require operational information, e.g., in the form of provenance [6]. Unfortunately, tuple-granularity provenance does not scale to the size of contemporary datasets, restricting provenance systems to coarse-grained information [8, 4].

While merging code without considering its effect on data is an option, it is error-prone. Thus, most merge conflicts need to be resolved manually [5]. In software development, manual conflict resolution regularly involves careful analysis, some guesswork and, crucially, good test coverage to ensure correctness [11]. Resolving conflicts in this fashion is infeasible for complex data-intensive pipelines with non-obvious semantics.

The problem is that existing tools separate code and data: merging data without the code is tedious, while merging code without data is risky. What is required is an approach that considers code and data when merging.

My group at Imperial College has started working on BOSS (Bulk-Oriented Symbol Store), a system that manages code and data in a single format — a concept known as Homoiconicity (popularized by Lisp [7]). Lisp-style Homoiconicity has two components: a structured view of the program code (nested lists of instructions in Lisp) and a means to manipulate it using compile-time functions (called macros in Lisp). This approach is useful, e.g., to implement domain-specific languages [3, 12]. In BOSS, we turn this concept around: the system allows storing value-producing “Expressions” (pieces of code in a Lisp-like language) in the database and evaluates them at query time. This idea generalizes User-Defined Functions, and also has many potential applications for data integration, cleaning, model management, and many more. For now, we focus on collaborative data science: BOSS stores the “operation graph” (depending on research field, one might call it the provenance or the version graph) of every data item directly in the tuple, enabling complex merges.

Three key challenges arise. For fully automatic merges, scalability is an unsolved problem. State-of-the-art tools (using either tuple-granularity provenance or the transaction log) can easily take hours or run out of memory, even at a moderate scale. To limit the effort of semi-automatic merging, users need diff/merge tools that operate in bulk, e.g., by exploiting provenance like “outliers, i.e., prices above £99 were clipped” or “prices were normalized to adjust for inflation” and letting the user select their order. Finally, a collaborative data science system needs to account for data manipulation by third-party libraries (Python, R, Tensorflow, etc.). Extracting provenance information from those will require whole-program analysis.
REFERENCES