Fast Queries Over Heterogeneous Data Through Engine Customization

Manos Karpathiotakis[†] Ioannis Alagiannis[†] Anastasia Ailamaki[†‡]

[†]Ecole Polytechnique Fédérale de Lausanne
{firstname.lastname}@epfl.ch

[‡]RAW Labs SA

ABSTRACT

Industry and academia are continuously becoming more data-driven and data-intensive, relying on the analysis of a wide variety of heterogeneous datasets to gain insights. The different data models and formats pose a significant challenge on performing analysis over a combination of diverse datasets. Serving all queries using a single, general-purpose query engine is slow. On the other hand, using a specialized engine for each heterogeneous dataset increases complexity: queries touching a combination of datasets require an integration layer over the different engines.

This paper presents a system design that natively supports heterogeneous data formats and also minimizes query execution times. For multi-format support, the design uses an expressive query algebra which enables operations over various data models. For minimal execution times, it uses a code generation mechanism to mimic the system and storage most appropriate to answer a query fast. We validate our design by building Proteus, a query engine which natively supports queries over CSV, JSON, and relational binary data, and which specializes itself to each query, dataset, and workload via code generation. Proteus outperforms state-of-the-art open-source and commercial systems on both synthetic and real-world workloads without being tied to a single data model or format, all while exposing users to a single query interface.

1. INTRODUCTION

The ongoing data explosion is leading to a major overhaul in a range of scientific and business domains. Practitioners have evolved into data scientists, relying heavily on data analysis over an increasing number of datasets. Besides relational tables, semi-structured hierarchical data formats have become the state of the art for data exchange. In addition, scientists use domain-specific formats and external structured files containing data modeled as tables, hierarchies, and/or arrays. Users execute widely different analysis tasks over all these data types. Heterogeneity, both in data and in query workload, significantly affects the way data analysis is performed.

Meaningful data analysis depends on combining information from numerous heterogeneous datasets: data-intensive domains such as sensor data management and decision support based on web clickstreams involve queries over data of varying models and formats. Users that want to perform analysis over heterogeneous datasets can use a database engine that supports multiple use cases, but this approach is expensive because such engines are typically overly generic and hard to optimize for all cases. Therefore, users typically settle for a dedicated, specialized system for each of their use cases [51]. Each of these two extremes either offers i) extensive functionality and expressiveness, or ii) minimizes response times in a particular scenario, but not both. Hence, performing analysis effortlessly and efficiently remains an open problem.

One proposed solution is to flatten the different datasets into the relational model and load them in an RDBMS [50]. Data types such as hierarchies, however, are not a natural fit for tables. Another alternative is the data federation of heterogeneous data sources [17, 23]. The dominant approach in this case is packaging together multiple query engines, using the appropriate one for each specialized scenario, and relying on a middleware layer to integrate data from different sources. Thus, besides the challenge of data integration, users face a system integration issue, which increases complexity. Alternately, data analysis frameworks [8, 53] keep data in a "data lake" regardless of its format. Native support for rich data models in these systems is typically limited because it complicates system architecture and query optimization. Queries over complex data therefore incur a performance penalty. An encompassing design choice of the previous approaches is that all datasets have to be fully ingested and converted into a default format per system, either as a pre-loading step or during query answering. This process adds an additional upfront cost per query. Finally, ViDa [33] envisions effortlessly abstracting data out of its form and manipulating it regardless of the way it is stored or structured. This is a promising direction, but ViDa only proposes an abstract system blueprint.

This paper presents a system design that bridges the conflicting requirements for generality in analysis and minimal response times. The design supports both relational as well as nested data by using an expressive, optimizable query algebra that is richer than the relational one. The algebra allows combining data of heterogeneous models and produces data-model-conscious query plans. We couple this powerful query algebra with on-demand adaptation techniques to eliminate numerous query execution overheads. Specifically, our design is modular, with each of the modules using a code generation mechanism to customize the overall system across a different axis. First, to overcome the complexity of the broad algebra, we avoid the use of general-purpose abstract operators. Instead, we dynamically create an optimized engine implementation per query using code generation. Second, to treat all supported data formats as native storage, we customize the data access layer of the system based on the underlying data at query time. Finally, to mimic

Proceedings of the VLDB Endowment, Vol. 9, No. 12

the storage that better fits the current workload, we materialize in-memory caches and treat them as an extra input. The shape of each cache is specified at query time, based on the types of data accessed and the query workload trends. Overall, the originally distinct modules collapse into a unified, specialized query engine at runtime.

We validate our design by building *Proteus*, an analytical query engine that queries heterogeneous datasets without converting them to a homogeneous form. Proteus couples a general query interface with the execution times of a system that has been specialized for a specific query, data, and workload instance. Proteus currently supports CSV, JSON, and relational binary data. Its modularity makes it extensible; adding support for more formats is straightforward.

Contributions. The contributions of this work are the following:

- We present a system design principle that offers i) generality in analysis and ii) minimal response times. To achieve this, the design couples i) a query algebra that supports both relational and nested data with ii) on-demand customization mechanisms that collapse all layers of the system architecture at query time. The final result is a highly-optimized specialized engine per query.
- Based on our design, we implement Proteus, a full-fledged analytical query engine that queries CSV, JSON, and relational binary data transparently and efficiently. Proteus uses code generation to specialize its entire architecture per query and to craft caching structures of different shapes to adapt to the workload.
- We show that Proteus outperforms state-of-the-art open-source and commercial solutions in a mix of workloads. We perform a fine-grained evaluation over TPC-H data using multiple data representations; Proteus performs as if it has been designed for each use case. We also execute a challenging real-world workload over a mix of diverse datasets, in which Proteus is $\sim 3\times$ to $9\times$ faster than the state-of-the art alternatives.

2. RELATED WORK

A large body of work proposes a variety of solutions for the problem of querying heterogeneous data and efficient query processing in general. This section surveys related work and highlights how Proteus pushes the state-of-the-art even further.

Data Federation. To cope with data heterogeneity, data federation approaches perform analysis over diverse data sources without placing all data in a single system [18, 20, 54]. In recent years, the dominant approach has become bundling together multiple systems, each with a different query engine, and using the most appropriate engine for each scenario. These *polystore* systems initially combined Hadoop with an RDBMS [3, 22]. Newer proposals [17, 23] bundle more engines to better fit more use cases. To treat multiple engines as one, the overall solution uses middleware to perform cross-system query optimization, query splitting, data exchange between systems, etc. Thus, besides data integration, system integration becomes a concern which complicates data analysis.

To address this concern, ViDa [33] envisions effortlessly abstracting data out of its form and manipulating it regardless of its structure. This paper advances the goals of ViDa by materializing a modular system design for queries over heterogeneous data. The distinct modules of the design collapse at query time, eventually resulting in a specialized implementation per query. We couple this architecture with ad hoc storage structures to adapt to the query workload. Finally, we evaluate design choices extensively over Proteus, a mature system implementation.

Raw Data Processing. Asking queries over raw data is an emerging paradigm [30]. NoDB [5] makes an RDBMS raw-data-aware. SDS/Q [13] and SCANRAW [21] perform parallel analysis over a scientific file format. RAW [32] generates code to make raw data accesses cheaper. Most systems focus on a specific format; RAW does discuss multiple formats, but flattens hierarchies because it is a relational engine. Instead, Proteus natively operates over various data models and formats, also generating an engine per query.

Native Engine Support. Commercial systems like System RX and XML DB are hybrids offering native support for both relational and XML data. System RX [11] uses XML-specific storage, an XQuery compiler, and XML indexes. XML DB [41] calibrates XML storage between CLOBs and objects "shredded" to rows. Oracle [38] and SAP [16] also discuss extending an RDBMS with a JSON datatype. The processing primitives of these approaches target particular formats (e.g., relations and XML), while Proteus customizes itself for a multitude of formats on demand; its operators are by design agnostic to the underlying data for extensibility.

Encoding Schemes. Various works advocate "shredding": flattening hierarchies and storing them in multiple tables [14, 50]. MonetDB [14] uses specialized data encodings, join methods, and storage for XML data. Argo [19] proposes similar encoding schemes for JSON. Shredding approaches pay a penalty to reconstruct complex objects because multiple joins are required to re-stitch an object. Finally, Sinew [52] and PostgreSQL use a custom binary serialization for JSON. Instead of fitting data to the query engine, Proteus specializes itself based on the data and query types. It operates natively over the original data instead of loading data using complex encodings. If needed, Proteus can materialize data subsets of interest into caches to emulate different encodings dynamically.

(SQL-on-)Hadoop & Cloud Systems. Multiple systems have been built over Hadoop or a similar distributed runtime environment to query heterogeneous datasets [1, 7, 8, 12, 44]. Jaql [12] and Pig Latin [44] are query languages for semi-structured nested data, and both get translated to MapReduce jobs. Spark SQL [8] introduces relational processing support over (semi-)structured data. Nested datatypes are again treated as objects that are opaque to the optimizer. Finally, Dremel [40] flattens nested data into columns.

Our work is applicable to the engines of these frameworks. For example, most of these systems use data serializers such as Avro to fully transform input datasets into a format they can process. Proteus, however, relies on input plug-ins that process only the data needed, and calls them at different steps of execution to judiciously convert input values, unnest nested structures, etc. Using plug-ins that are tightly integrated with the rest of the engine instead of "black boxes" that blindly ingest data can benefit these systems.

Code Generation. Runtime code generation has become an established mechanism, used by several relational engines [6, 34, 36, 43, 45, 49]. HIQUE [36] generates cache-conscious code via code templates. HyPer [43] uses the LLVM compiler [37] to generate low-level machine code. LegoBase [34] goes through numerous rewriting ("staging") steps to generate C code. Proteus follows the HyPer paradigm and relies on LLVM too. Proteus is more expressive than relational code-generated engines because it supports multiple data models and transformations between them. Moreover, Proteus treats each supported data format as its native storage and adapts to incoming queries better because it makes dynamic decisions about its data access mechanisms, "tuple" structure, and cache organization, all of which are predefined in other systems.

3. AN EXPRESSIVE QUERY ALGEBRA

We want to enable queries over a multitude of data models, hiding the underlying heterogeneity. Thus, our query algebra must treat all supported data types as first-class objects in terms of both expressive power and optimization capabilities, instead of considering richer types as BLOB-like values which are opaque to the query optimizer. Existing approaches follow two main directions

Operator Name	Select	(Outer) Join	Reduce	Nest	(Outer) Unnest
Operator Symbol	$\sigma_p(X)$	$X \rtimes_p Y$ $X \bowtie_p Y$	$\Delta_p^{\oplus/e}$	$\Gamma_{p/g}^{\oplus/e/f}$	$\not\mu_p^{path}(X)$ $\mu_p^{path}(X)$
Superscript & Subscript	p: Filtering Expression f: Groupby Expression *path*: Field to unnest		e: Output Expression g: Non-nullable Expression $\oplus$: Output Collection/Aggregate		

Table 1: The operators of the nested relational algebra.

to deal with the data model variety. Each of them, however, sacrifices either generality or query performance.

The first approach involves building an entire system with a specific data model in mind and specialized to the use case at hand. A prominent example is the use of column-oriented DBMS for analytical relational workloads. Following the same trend, systems like CouchDB and MongoDB emerged for semi-structured data. Given that they are optimized for non-relational cases, they impose a number of restrictions for more "traditional", relational-like workloads. For example, data entries are assumed to be de-normalized as self-contained objects, so joins are challenging to express. Because each specialized system supports only a specific type of input efficiently, users resort to system integration, i.e., having a dedicated system for each of their dataset types and using a mediation layer over them to handle cross-dataset queries.

The second approach is to extend an established system with support for additional data types, e.g., adding support for JSON to an RDBMS. The extension is typically inefficient: A proper extension would add explicit query operators to support the new types of data, which requires significant engineering effort, as well as extending the (relational) model to which every system component adheres. Due to these constraints, commonly only functions that access and manipulate the new complex data are introduced, and the system's optimizer remains unaware of the new data type particularities.

We use a third, different approach to allow queries across data of various models: We leverage a unifying data model and a powerful query language internally. Proteus is built around the *monoid comprehension calculus* [24] because this calculus supports various data collections (e.g., bags, sets, lists, arrays) and arbitrary nestings of them. The monoid calculus and its corresponding algebra are optimizable and allow transformations across data models, hence Proteus can produce multiple types of output. The calculus is also expressive enough for other query languages to be mapped to it as syntactic sugar: For relational queries over flat data (e.g., binary and CSV files), Proteus supports SQL statements, which it desugarizes to comprehensions. For more powerful manipulations of flat data (e.g., outputting results that contain nestings) and for queries over datasets containing hierarchies and nested collections (e.g., JSON arrays), Proteus currently exposes a query comprehension syntax to the user; Example 3.1 presents a query using this syntax.

For each incoming query, the first step is translating it to a calculus expression. The calculus expression is then rewritten to an algebraic tree of a *nested relational algebra* [24]. This algebra resembles the relational one, and relational optimization techniques are applicable to it. On top of that, it offers first-class support for operations related to unnesting of queries over nested data. The operators of the nested relational algebra are depicted in Table 1. The *selection*, *join*, and *outer join* operators are identical to their relational counterparts. *Reduce* and *nest* are overloaded versions of the relational projection and the grouping operator respectively. Finally, the *unnest* and *outer unnest* operators "unroll" a collection field *path* that is nested within an object.

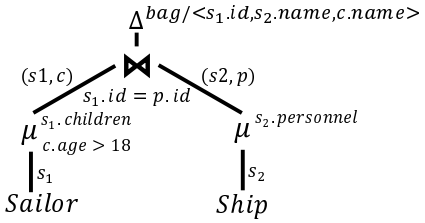

Figure 1: Query involving unnest operators: Without them, the operators higher in the tree would have to process BLOBs repeatedly every time they need a nested value.

Example 3.1: Suppose we have a dataset comprising sailors and a dataset comprising ships. Each sailor has an *id* field and a *children* field which contains a list of *(name,age)* pairs for the sailor's children. Each ship entry has a *name* field and a *personnel* field which contains a list of sailor identifiers. The query "For each Sailor, return his id, the name of the Ship on which he works, and the names of his adult children" is expressed in the calculus as follows:

```
for { s1 <- Sailor, c <- s1.children, s2 <- Ship,
      p <- s2.personnel, s1.id = p.id, c.age > 18 }
      yield bag (s1.id, s2.name, c.name)
```

The resulting plan for this query is depicted in Figure 1. Two unnest operators handle explicitly the nestings in the data.

Overcoming Complexity. Using a rich data model and language/algebra for queries over complex data was proposed when OODBs and XML appeared [24, 25, 55]. Rich models and algebras, however, lost traction due to their complexity. The more complex an algebra is, the harder it becomes to evaluate queries efficiently: Dealing with complex data leads to complex operators, sophisticated yet inefficient storage layouts, and costly pointer chasing during query evaluation. To overcome all previous limitations, we couple a broad algebra with on-demand customization.

4. THE ARCHITECTURE OF PROTEUS

Proteus is a query engine designed from scratch to enable fast queries over heterogeneous datasets. To provide generality, Proteus uses an algebra which can model operations across different types of data, thus offering expressive power and rewriting opportunities for queries targeting complex data. To also minimize response time, Proteus creates a new query engine instantiation on-demand per query via code generation. Furthermore, Proteus customizes its storage structures to adapt them to the workload. The result is a custom, highly-optimized engine, expressed in machine code and operating over a data representation that suits user analysis.

Figure 2 depicts the components of Proteus. The *Query Parser* handles incoming queries, which are then rewritten to a physical plan by the *Query Optimizer*. *Algebraic Operators* encapsulate data model heterogeneity; they express the plan of a query and coordinate code generation. *Expression Generators* generate code for expression evaluation when requested by an operator. *Input Plug-ins* encapsulate data format heterogeneity; they consider source-specific optimizations and generate code that accesses any required data. They also provide statistics and costing formulas per data source. *Output Plug-ins* generate code that handles operator output and cache creation along with the *Memory & Caching Managers*.

Query Optimization. Systems which process heterogeneous data face the following challenges: First, queries over hierarchical data typically involve many levels of nesting, which increases execution overheads. Second, unless an optimizer has access to data statistics, it may produce suboptimal plans. Proteus uses a three-step approach to address these issues: First, when a user asks a query, Proteus parses and normalizes it, performing operations

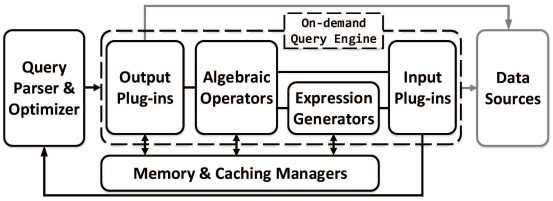

Figure 2: Proteus architecture.

such as selection pushdown and unnesting multiple types of nested queries. Then, Proteus rewrites the query to a nested relational algebra. The algebraic representation is amenable to relational-like optimizations and further unnesting. Finally, after a number of rule-based rewrites, the optimizer considers cost-based transformations; it follows a bottom-up strategy and relies on gathered statistics to perform access path selection and join re-ordering. Its difference from traditional optimizers is that statistics and costing of data accesses are provided by the input plug-ins relevant per query.

On-demand Query Engine. The operators of traditional query engines are hard-coded to a database-specific input data format for efficiency. Proteus is designed to treat each data format as native storage. To cope with data heterogeneity, Proteus masks data complexity from the operators by using an input plug-in per data format. Each plug-in exposes a uniform interface that the rest of the engine uses to consume data values. The algebraic operators process input either by calling expression generators or via direct interaction with an input plug-in. This separation of concerns makes Proteus extensible: adding a plug-in suffices to support a new data format.

The operators of Proteus call output plug-ins to handle the creation of output and the materialization of any required intermediate results during query execution. Proteus also uses the output plug-ins to define caching structures, which it populates as a side-effect of execution to adapt to the overall workload. Once materialized, Proteus treats caches as an additional input dataset.

For each query, Proteus uses a code generation mechanism to collapse the layered architecture of the engine – the dashed part of Figure 2 – into a specialized piece of code. Each of the components produces low-level machine code that Proteus combines to form a program serving the currently processed query. Specifically, once the optimizer has produced a physical plan, Proteus traverses it recursively until it finds the datasets to access (i.e., the leaf nodes). It then triggers the appropriate input plug-ins to generate code accessing data. As the recursion is returning control to the root node of the plan, Proteus generates code for every visited operator. Each visited operator may (re-)trigger input and output plug-ins to process its input and/or materialize its output.

Memory Manager. Whenever a system component requests a memory block to read/write, the Memory Manager handles the request. The Manager distinguishes between input files and caching structures: It memory-maps input files, treating all input data as if it is memory-resident, and delegates paging to the OS virtual memory manager. As for caching structures, Proteus pins them in a memory arena, and uses an LRU variation to evict them when appropriate.

Implementation Scope. Proteus targets read-only and append-like analytical workloads. This study focuses on full dataset scans and excludes value-based indexes, as they are an orthogonal optimization which is straightforward – yet optional [10] – to implement. In case of data updates, Proteus currently drops and rebuilds any affected parts of existing auxiliary structures (e.g., caches).

5. ON-DEMAND QUERY ENGINES

Ideally, a system must allow diverse queries over heterogeneous datasets, but also perform as if it has been designed for a specific use case – even better, as if it is hard-coded to serve a specific query: For analytical queries over flat (e.g., binary, CSV) data, the system must be as fast as an analytical relational engine. For hierarchical data, it must be as fast as a document store.

The nested relational algebra of Proteus enables querying complex data types and considers them as first-class citizens during query optimization. It also facilitates query unnesting – a common issue when input data is nested. Dealing with complex data and query operators, however, comes at increased cost.

Even when dealing with the strictly relational operators of an RDBMS, interpreting the query plan is costly. A source of overhead is the ubiquitous Volcano iterator model [27], which enables pipelining and exposes a single interface for all operators, but complicates control flow and introduces multiple function calls per tuple processed (e.g., each operator calling *getNextTuple()*). Another factor is the variety of datatypes that each operator must be able to process: An operator must trigger different code paths depending on whether its arguments are i) integers, ii) floats, iii) some combination, etc. To support this behavior, operators use control flow statements and (virtual) function calls in their code, which leads to increased branching in the critical path of execution.

This *interpretation overhead* [36, 43], stemming from function calls and control flow statements that disrupt the instruction pipeline, affects pipelined query execution negatively. Intuitively, the nested relational algebra operators face similar issues. Even worse, they have to i) support additional, more complex types of input, and ii) perform extra work compared to their relational counterparts. For example, besides the selection and join operators, many additional operators of the nested relational algebra have an embedded filtering step (e.g., unnest, reduce). The additional complexity further increases the interpretation overhead.

One way to remove the interpretation overhead is to use a block-oriented, operator-at-a-time execution model, as columnar engines typically do [15]. The block-oriented model, however, introduces materialization overhead per operator. This cost would be more severe for Proteus compared to traditional relational systems because of the more complex datatypes to be materialized. Even worse, Proteus serves datasets whose contents rarely reside in explicit data blocks, so every query would pay an upfront cost to materialize input blocks. Instead of processing data blocks, Proteus pipelines data through its operators, but also minimizes interpretation overhead by customizing itself when it receives a query based on **i)** the query requirements and **ii)** the datasets the query touches.

5.1 An Engine per Query

Traditional pipelined query engines execute a query by interpreting its physical plan and invoking multiple general-purpose operators for each input tuple. Proteus removes interpretation overhead by traversing the query plan only once and generating a custom implementation of every visited operator. Proteus thus uses control flow mechanisms such as datatype checks only during the single traversal, and avoids the per-tuple penalty that a static pipelined engine incurs. Once all plan operators have been visited, Proteus blends the generated code stubs into a hard-coded query engine implementation which is expressed in machine code.

Proteus uses LLVM [37] to generate low-level code, which it compiles at runtime. LLVM is a collection of compiler infrastructure which offers frontends for languages such as C/C++ and Fortran. In its core, LLVM translates these languages into an intermediate representation (IR) resembling assembly code: the *LLVM IR*. LLVM then compiles the IR into actual machine code based on the underlying hardware. Proteus generates LLVM IR because i) it is strictly-typed and less error-prone than macro-based C++

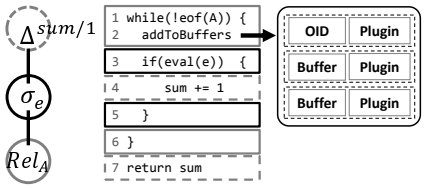

Figure 3: Example of a query plan and of the generated (pseudo-) code. Once the scan operator places needed fields in virtual buffers, they are used to evaluate the filtering expression.

code, ii) it compiles much faster than macro-based C++ code, and iii) LLVM offers rewrite passes such as dead code elimination that optimize the generated IR. In summary, Proteus uses LLVM as a plan rewriting mechanism, and performs one extra step compared to traditional query engines: It rewrites the physical algebraic plan – an abstract, high-level IR – into the imperative, low-level LLVM IR which is amenable to compiler-centric optimizations [2].

After parsing and optimizing a query, Proteus traverses the physical plan of the query in post-order depth-first-search (DFS). When visiting a node of the plan, Proteus i) visits the node's children to produce the code corresponding to their functionality, ii) generates the physical implementation corresponding to the current node, and iii) returns control to the node's parent to continue the code generation process. The recursive traversal terminates when it reaches a leaf node (a scan operator). Proteus then generates a code stub that, when executed, will launch a scan over a dataset. In each scan iteration, the generated code will access a "record" from the data and place the fields needed for the rest of the plan in virtual memory buffers. The virtual buffers can be thought of as local variables placed in the stack frame. To maximize locality, the LLVM compiler promotes buffer contents to CPU registers when possible. Therefore, subsequent operators referencing values that exist in register-backed buffers experience minimal access times and fully pipeline data. Once Proteus has generated code stubs for a leaf node, it shifts control to the node's parent, also passing along pointers to the virtual buffers and to the currently "hollow" parts of the overall query code that need to be filled in next. The same process continues until control returns to the root node.

Figure 3 depicts a plan for the query `SELECT COUNT(*) FROM A WHERE e`, along with a high-level description of the resulting code. The scan of relation A results in the generation of a "hollow" while-loop. The code for the ending condition of the loop (line 1), as well as for populating virtual buffers with the fields necessary to answer the query (line 2), is injected by an *input plug-in* that allows the data-format-agnostic scan operator to interface with dataset A regardless of how it is stored. Then, the selection operator generates a hollow *if* block, whose outcome depends on the evaluation of the expression *e* (line 3) in each iteration. Proteus retrieves the values required to evaluate *e* from the virtual buffers. The reduce operator calculates the final result by incrementing a counter, which it then outputs. The result of the physical plan traversal is not a number of standalone operator implementations: It is a minimal, specialized piece of code representing an entire query, with operator logic tightly stitched together to ensure pipelined query execution. This type of execution minimizes intermediate query results, maximizes code and data locality, and reduces register pressure.

Proteus also uses pre-existing (i.e., not generated) C++ code for some of its functionality. Proteus wraps these operations in C++ functions and calls them when appropriate from the generated code. For example, the Memory and Caching Managers do not generate code. In another case, Proteus uses hash-based algorithms for the join and grouping operators, namely variations of the radix hash join algorithm [39] adapted from [9]. While parts of the join implementation are indeed generated at runtime, other parts, like clustering the materialized entries based on their hash values, are wrapped in a C++ function. This function is only called once per join side, so the overhead of making the function call is minimal.

Input Plug-in Methods		
generate()	hashValue()	unnestInit()
readValue()	flushValue()	unnestHasNext()
readPath()		unnestGetNext()

Table 2: The input plug-in API of Proteus

Implementation. The layers of Proteus that parse, rewrite, and optimize queries are expressed in Scala and output a physical query plan. The layers that traverse the query plan and trigger code generation using LLVM are written in C++. When Proteus receives a query, it generates stubs of LLVM IR, which it stitches together during the traversal of the physical query plan and puts into a single function. Within milliseconds, LLVM compiles the IR of the function into actual machine code based on the underlying hardware. The result is a library which Proteus calls to serve the query.

5.2 A Custom Data Access Layer per Query

The operators of Proteus access a dataset either by triggering an expression generator to produce code for the evaluation of an algebraic expression, or by directly calling the corresponding input plug-in. This separation of concerns ensures extensibility.

Expression Generation

Proteus places values from each dataset it touches into virtual memory buffers, which the query operators use to evaluate expressions of the nested relational algebra. For example, if Proteus has populated buffers with fields *a.sal* and *a.bonus*, it can evaluate the filtering expression of the operator $\sigma_{sal+bonus<3000}$. The physical operators assign the evaluation of algebraic expressions to an expression generator. In the example of Figure 3, an expression generator produces the code to calculate the result of $eval(e)$ at line 3, and injects it as the condition in the *if* statement. Similar generators are used when hashing an expression and when flushing out the query output. A useful property of this separation is that the operators are agnostic to the underlying data models/formats/properties. The operators are oblivious to whether a value in the memory buffers belongs to an array, is nested, or is not fully materialized yet; all they require is that the expression generators inject the appropriate code for expression evaluation at the code spots they designate.

Input Plug-ins

Proteus masks the details of the underlying data values from the query operators and the expression generators. To interpret data values and generate code evaluating algebraic expressions, Proteus uses *input plug-ins*. Each input plug-in is responsible for generating data access primitives for a specific file format. Proteus currently uses input plug-ins for CSV, JSON, and relational binary data (both row-oriented or column-oriented).

Table 2 lists the API that every input plug-in exposes. Calls to a plug-in can be made by i) a scan operator populating virtual memory buffers (the *generate()* call), ii) an unnest operator looping through a nested collection (*unnestInit()* etc.), or iii) an expression generator calculating an expression. In the third case, *readValue()* provides a field's value to the expression generator, and *readPath()* returns a pointer to a data object's field. Consecutive calls to *readPath()* are used to access nested fields.

When a scan operator calls an input plug-in, the plug-in generates code that customizes the data access layer of Proteus based on **i)** the *current query requirements* and **ii)** the characteristics of the

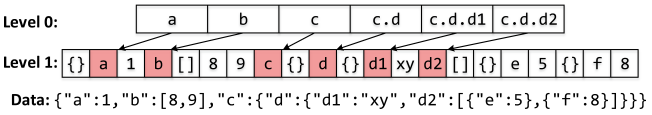

Figure 4: Example of a structural index for a JSON object.

dataset to be accessed: its *schema*, *format*, and *contents*. Using this information, Proteus generates code that performs fewer and more efficient data accesses than a general-purpose scan operator. An example of exploiting the query requirements is the following: During query rewriting, Proteus pushes field projections down to the scan operators so that it pays to extract only the fields necessary. To perform these selective accesses, a general-purpose scan operator would use a loop that checks whether each field is needed for the query, thus introducing branches in the critical path of execution. Instead, Proteus generates code processing only the required data fields. Proteus also uses the dataset schema to avoid unnecessary control logic such as datatype checks – it generates specific access primitives for integer fields, nested fields, etc. The overall code generated for scanning data resembles a hard-coded program.

Specializing per Dataset Format. Proteus generates code that considers the particularities of each data format. For binary relational data, an input plug-in generates code reading the memory positions of the required data fields. For more verbose or richer formats, Proteus uses more sophisticated access methods. The common denominator of all input plug-ins is that for every data object / "tuple" they access, they produce an object identifier (*OID*), which they forward to the query operators. As an example, for flat data the OID is a row counter. Using an entry's OID, an expression generator can invoke the corresponding input plug-in at a later point in execution to access a value needed for an expression's evaluation.

Apart from creating an OID, Proteus calibrates how eager / lazy the generated access primitives are (i.e., which values to place in memory buffers apart from the OID, whether to eagerly convert a value to a binary serialization, etc.). Proteus supports lazy plug-in behavior because eagerly populating memory buffers may prove unnecessarily expensive. When performing a path query over nested objects or data unnesting, Proteus avoids eagerly serializing a complex object only to process a subset of it: Instead, Proteus uses structural information for the data in order to navigate in the dataset and to access only the values necessary to provide a result. In addition, in many cases Proteus delays data conversion because it may prove to be unnecessary (e.g., because of some selection filtering out results). Another scenario is applying different materialization policies in relational workloads. To allow for this flexible behavior and enable a field's reconstruction at any point, Proteus maintains plug-in information for each field value in its memory buffers. For every such value, the corresponding plug-in uses rules to specify how lazily to process it based on criteria such as its data type and at which point of the query it is used, and generates appropriate code.

Structural Indexes. The input plug-ins of Proteus use auxiliary structures to reduce the navigation cost associated with verbose data formats, for which every access requires substantial parsing effort. These *structural indexes* store *positional* information about fields in the datasets instead of actual data values. Their entries are addressable by OID, so that all plug-ins have uniform behavior. For CSV datasets, structural indexes store the binary positions of a number of data columns in each row [5]. Proteus stores the position of every Nth field of the file (e.g., if N=10, it stores the positions of the 1st, 11th, ... fields). When looking for a field, Proteus locates the closest indexed field position and starts seeking from that point.

Structural indexes for JSON require a more involved process because of the inherent complexity of the JSON format, which allows arbitrary levels of nesting and field order, as well as optional fields. When Proteus accesses a JSON file for the first time, it validates the JSON input. During validation, Proteus populates an index per JSON object with structural information. The resulting structural index serves two goals: It reduces the parsing effort for subsequent accesses to the file, and minimizes the interpretation overhead stemming from the schema flexibility of JSON data.

Each entry of a JSON structural index captures information about a *token* (e.g., a field name, an array, etc.) contained in a JSON object: its binary *starting and ending positions* in the file, as well as its *type*. To serve requests for a data field, the JSON input plug-in finds its corresponding token entry in the structural index. The plug-in then forwards the entry identifier – which acts as an OID – to the operator / expression generator that requested it.

The structural index described so far corresponds to "Level 1" of the example in Figure 4. The first index entry, labeled "{}", keeps the starting and ending positions of the overall JSON object, the second entry keeps the positions of token a, and so on. Intuitively, if a dataset contained multiple objects similar to the one depicted and a query requested field a from each one, the JSON plug-in would follow the same process for each object: Return the second entry of the object's corresponding structural index. Nevertheless, there is no guarantee that field a comes before field b, b before c, etc. in every object of the dataset. Thus, Proteus would have to sequentially scan each object's index and compare the label of the wanted field with the one currently visited.

To remove this overhead, Proteus introduces "Level 0" to the structural index. "Level 0" comprises an associative array which maps field names to their corresponding positions in "Level 1" of the index. The shaded values in Level 1 are now redundant and thus removed. The JSON plug-in finds a field's position via a lookup to the associative array. The use of Level 0 reduces data access costs and offers determinism despite the semi-structured JSON nature.

Proteus also registers nested records in Level 0. In Figure 4, by storing a pointer to field $c.d.d1$, dereferencing occurs in one step instead of multiple ones. Nested collections are treated otherwise: Notice that fields e and f, which correspond to the contents of a nested (array) collection, are omitted from Level 0. JSON structural indexes opt against maintaining pointers to array contents because Proteus has an explicit *Unnest* operator to handle nested collections. The code path of Unnest applies the same action to every nested element, therefore it is unaffected by schema flexibility.

Specializing per Dataset Contents. The more information Proteus obtains about a dataset, the more aggressive optimizations it performs: An input plug-in can craft an optimized code path suitable only for a specific file instance. In the case of JSON data, schema flexibility introduces overhead even when using a structural index, because Proteus has to store more bookkeeping information per indexed entry, and generate more complex code to process it. There are many scenarios, however, such as the case of machine-generated data, where every object in a dataset has the same fields in the same order. Proteus can verify whether this case holds while creating a structural index, and drop Level 0 because the lookup process is now deterministic: It is sufficient to maintain the sizes of any variable-length structures (i.e., JSON arrays) met and combine them with fixed schema information to deterministically compute the exact positions of relevant structural index entries. The result is a more compressed structural index and an efficient code path for lookups. In a similar optimization, if a CSV file contains fixed-length entries, Proteus deterministically computes field positions and injects them in the code instead of using a structural index.

Enabling Cost-based Optimizations. Proteus uses a metadata store to maintain statistics per data source, namely dataset cardi-

nalities and min/max values per attribute, and delegates statistics collection to each input plug-in. The statistics collection process is fine-tuned to avoid introducing execution overheads. Specifically, Proteus refrains from generating code for statistics gathering in every query to avoid bloating the minimal generated code. Instead, it collects statistics in three ways: First, Proteus collects statistics during the first (cold) access to a dataset, because I/O masks the overhead of statistics gathering. Second, when the plan contains a blocking operator (e.g., a join), the relevant input plug-in injects code that profiles the materialized values all at once. Finally, a daemon process periodically triggers statistics-gathering queries when the system is idle – a methodology followed by multiple DBMS. Regarding costing, each input plug-in uses different cost formulas, which it instantiates with data statistics to provide cost estimates to the query optimizer. Delegating source-specific work to a "wrapper" per source is also popular with federated systems [46, 47].

Proteus allows plug-in developers to calibrate statistics gathering and costing. The baseline option is to use predefined, hard-coded estimates in place of statistics-based computations (e.g., assume that the default selectivity of a predicate is 10%), as well as textbook cost formulas. Proteus offers such a skeleton for every input plug-in by default because it has been shown to have satisfying results [46]. Regarding statistics, Proteus allows developers to adjust/change the sampling function to be called during cold queries and result materialization. The function specifies the type of sampling to be used, and on which fields the statistics-gathering mechanism should focus on. Regarding costing, the developer can change the provided cost formulas with more suitable ones for her needs.

Adding More Inputs. Adding support for more inputs is straightforward. For each new input, what is required is to code in an input plug-in which implements the methods of Table 2. A developer can use plain C++ instead of the lower-level LLVM API, since Proteus can directly call C++ functions. Of course, using the LLVM API leads to minimal, more efficient code; the plug-in developers decide how to calibrate ease of development and high performance based on their requirements. The same trade-off applies when integrating Proteus with existing data stores such as an RDBMS: A plug-in can either i) issue SQL queries to the DBMS, or ii) directly access the proprietary binary data format that the DBMS uses internally.

6. ADAPTING STORAGE TO WORKLOAD

Proteus dynamically populates data caches as a side-effect of query execution to adapt to the workload trends. These caches can be viewed as dynamic materialized views [35], following the data recycling principle [31, 42] of automatically caching results during query evaluation for possible reuse in the future. Proteus deals with complex models and formats, so the importance of reuse is even higher because of the effort needed to re-access the data involved and recompute the expressions that queries require. Since users express a range of queries over a variety of data, the caches must facilitate each diverse workload, adapting to serve it efficiently. Therefore, instead of having a predefined structure, the caches adapt to the types of queries asked. Depending on the query workload, the caching structures can resemble i) pages filled with tuples in a system's buffer pool, ii) binary columns accessed by a columnar engine, iii) nested objects serialized in a binary format, etc.

Proteus can cache any expression supported by the nested relational algebra. Each query may trigger the population of caches of different shapes – caches of different shapes can even be built at different phases of the same query. For example, a query subtree processing hierarchical data may benefit from a different cache type than the query part touching relational tables. Some expression types that Proteus can cache are the following:

- Field projections ($rel.attrA$).
- Arithmetic expressions ($(rel1.salary + rel2.bonus) * 12$).
- New record constructions ($< rel.attrA, tree.attrB.attrB1 >$).

Proteus uses caching primarily to benefit queries over non-binary, verbose sources such as CSV. By caching data entries in a more compact binary format, neither parsing nor data conversions are required to access them. Caching is also beneficial when a different data layout is more suitable for the workload than the one currently used by a dataset [6, 28]. Proteus is flexible enough to allow different caching policies depending on the expected workload type.

Implementation. The algebraic operators are oblivious to which expressions are to be cached and which of the input values they process is actually served from caches. When Proteus has to materialize data (e.g., during a join), or the physical plan contains a caching operator, Proteus assigns the task to an *output plug-in* that specifies i) the expression to be cached, ii) what the serialization format will be, and iii) the "degree of eagerness" to be used during caching. For example, when dealing with variable-length string entries, it might be sufficient to cache their binary starting positions, or even the OID of the entry to which they belong. Different types of workloads benefit from different policies across these axes.

Output plug-ins trigger cache construction similarly to expression evaluation: For each data entry, an expression generator produces code which evaluates the expression to be cached and places the result in a consecutive memory block. Proteus exposes the data cache as an additional input. As with the rest of the datasets, Proteus accesses the cached data using a dedicated input plug-in.

Building Caches. Proteus triggers cache creation i) *implicitly*, as a by-product of an operator's work, or ii) *explicitly*, by introducing caching operators in the query plan. *Implicit* caching exploits that some Proteus operators materialize their inputs: nest and join are blocking and do not pipeline data. Especially for joins, Proteus uses a radix hash-join variation, which materializes both input sides. It is thus important to re-use populated data structures and avoid rebuilding them, especially if the data originated in a verbose data format for which accesses are expensive.

For *explicit* caching, Proteus can place buffering operators at any point in the query plan. An explicit caching operator calls an output plug-in to populate a memory block with data. Then, it passes control to its parent operator. Creating a cache adds an overhead to the current query, but it can also benefit the overall query workload: When accessing verbose formats like JSON, it is advisable to avoid re-accessing the original data whenever possible. Even when using auxiliary structures to navigate in the file, there are still additional costs. After locating a required field, the input plug-in typically needs to convert it to a binary form. In addition, verbose objects pollute CPU caches with unneeded information. Each field that Proteus needs is located at an arbitrary position in the file. When Proteus places it in a CPU cache line, the rest of the line is typically filled with an unneeded part of the overall JSON object. Dealing with compact, packed binary caches greatly improves data locality. Therefore, if a cached field ends up being re-used, the benefit from avoiding these data accesses and computations is significant.

Cache Matching. For every cache that Proteus populates, the Caching Manager stores the physical plan corresponding to the cache and uses it as a search key during cache matching. Proteus considers the available caches right before generating code for a query. The cache/view matching process resembles that of [42, 48]. Proteus treats the physical plan as a DAG, where each node corresponds to a physical operator, and traverses it in bottom-up fashion. The Caching Manager traverses each stored plan simultaneously with the traversal of the current plan. For every node of the DAG visited, Proteus probes the Manager for nodes in the cached plans

that can be used instead. For a node in the current query to *fully match* a node in a cached plan, i) they must both perform the same operation (e.g., selection), ii) have the same arguments (i.e., evaluate the same algebraic expressions), and iii) their children nodes must match each other respectively. Whenever the Manager finds a match, Proteus applies the same process recursively until it reaches the root of a cached plan. If successful, Proteus rewrites the plan to use the cache. Besides full matches, Proteus considers *partial matching*: If Proteus has cached the intermediate results (i.e., the hash tables) of $A \bowtie B$, then the newly arrived query $A \bowtie C$ can reuse the hashtable built for A if it uses the same join key. Future work includes adding support for *subsumption* [26, 48], i.e, identifying that the cached tree $\sigma_{x>0}(A)$ can replace the current sub-tree $\sigma_{x>10}(A)$ as long as we re-apply the $x > 10$ predicate.

In summary, rewriting scenarios include replacing i) a sub-tree of the plan (e.g., a scan and a subsequent unnest operator), ii) a single operator (e.g., a scan), or iii) a part of an operator (e.g., one of the already materialized sides of a radix hash join). Code generation is an enabler for such rewrites of varying granularity, because it allows Proteus to generate code only for the necessary operations.

Cache Policies. Selecting which views to materialize is a well-studied research problem [29]. Proteus applies different materialization policies depending on the workload characteristics. Proteus benefits significantly when it places caching operators close to the leaf nodes of the plan in order to convert input (raw) values to a binary format. A reason is that raw data access is a major overhead when querying heterogeneous datasets. In addition, the simpler an operator tree corresponding to a materialized result is, the more upcoming queries will be able to re-use it and benefit from it. Therefore, the Caching Manager currently focuses on ways to fully replace a costly access path instead of materializing the result of a complex query sub-tree; applying more sophisticated policies and studying their effect [4, 29] is part of our future work. Proteus thus opts for straightforward first-come-first-served caching policies, and eagerly caches values read from CSV and JSON files. Proteus caches primitive values found in files containing hierarchies to avoid re-navigating through them, especially if the involved objects are deeply nested. Proteus also caches fields used as filtering predicates. On the contrary, Proteus avoids caching variable-length string fields from CSV and JSON files, which may be verbose and pollute the caches. Regarding cache eviction, Proteus uses a data-format-biased version of LRU, favoring data from inputs that are more costly to access (where $JSON \gg CSV \gg Binary$).

7. EXPERIMENTAL EVALUATION

We evaluate Proteus using i) synthetic benchmarks to isolate the performance of common query operations, and ii) a challenging real-life spam email analysis workload provided by Symantec.

Experimental Setup. We compare Proteus against a) systems that at some point were extended to support richer data models, and b) systems specialized for a specific scenario by design. Specifically, we compare i) PostgreSQL 9.4.1, ii) commercial DBMS X, iii) MonetDB 11.19.9, iv) commercial DBMS C, and v) MongoDB 3.0.3. PostgreSQL and DBMS X are row stores that support both relational and JSON data; they showcase how a generic system performs in the two diverse cases. We configure DBMS X to use its "main memory accelerator", therefore it keeps data in memory using a custom memory-friendly layout. MonetDB and DBMS C are read-optimized column stores, designed to efficiently support relational analytical queries, which recently added JSON support. Finally, MongoDB is a specialized system for JSON data, for which it uses a binary serialization (BSON). PostgreSQL supports both a binary (jsonb) and a character-based JSON serialization; we use jsonb because of its efficiency. The other systems treat JSON as a subtype of VARCHAR. Neither the systems we compare against nor Proteus make assumptions about field order in the JSON files.

We run all experiments on a dual socket Xeon Haswell CPU E5-2650L (12 cores per socket @ 1.80 GHz), equipped with 64 KB L1 cache and 256 KB L2 cache per core, 30 MB L3 cache shared, 256 GB RAM, and 2TB 7200 RPM SATA 3 disk storage. The operating system is Red Hat Enterprise Linux 7.1. Proteus uses LLVM 3.4 to generate custom code with the compilation time being at most $\sim$ 50 ms per query. We run all systems in single-threaded mode.

7.1 Specializing the Query Engine on Demand

This experiment isolates the performance of typical query operations over both hierarchies and relations. We use JSON and relational binary data, and examine a range of query templates with 10%, 20%, 50%, and 100% selectivity.

We use the TPC-H **lineitem** and **order** tables as input, using scale factors 10 (**SF10** - 60M lineitem tuples, 15M order tuples) and 100 (**SF100** - 600M lineitem tuples, 150M order tuples). We shuffle each file's contents to avoid potential optimizations that exploit interesting orders and can introduce noise to the experiments. To test performance over JSON data, we convert the TPCH-SF10 tables into a 20GB JSON file for lineitems and a 3.5GB file for orders, and load them in all the systems we compare against. As an indication of storage size, PostgreSQL requires 27GB to store the JSON version of lineitem, and MongoDB requires 30GB. Proteus natively operates over the JSON files and builds a structural index during the first data access. Index size is $\sim$ 21% of the JSON file for lineitems and $\sim$ 15% for orders, and its construction is significantly faster than loading the data in the other systems (e.g., $\sim 4\times$ faster than MongoDB). For experiments over binary data, we load the TPCH-SF100 version in PostgreSQL, DBMS X, MonetDB, and DBMS C. Proteus operates over binary column files similar to the ones of MonetDB. All systems operate over warm OS caches. Unless otherwise specified, the adaptive caching of Proteus is deactivated. The data types are numeric fields (integers and floats).

Projections. For queries projecting a varying number of fields, we use three variations of the following query template: `SELECT AGG(val1),...,AGG(valN) FROM lineitem WHERE l_orderkey < [X]` . The first two variations compute COUNT and MAX respectively. The third variation computes four aggregations (COUNT and MAX).

Figure 5 plots results for the JSON version (SF10). Proteus is the fastest system because its lightweight generated code path makes it more efficient for the CPU-intensive task of processing JSON entries. In addition, contrarily to PostgreSQL, Proteus does not treat JSON objects as bulky BLOB data; it uses the structural index to retrieve the information it needs from each object, which it then feeds in the query pipeline without "polluting" the CPU caches with the verbose JSON object any further. As for the other systems, JSON access is expensive for DBMS X because it uses a character-based encoding. MongoDB is competitive with PostgreSQL only for the COUNT query. As the number of aggregates to compute increases, PostgreSQL outperforms MongoDB. JSON support is still immature in MonetDB, which results in suboptimal performance. Similarly, DBMS C underperforms in all our experiments over JSON data. For this reason, and because some of the benchmarked operators are either work-in-progress (e.g., unnest) or not yet supported efficiently (e.g., using a JSON field in a GROUP BY clause requires a costly workaround for MonetDB), we exclude MonetDB and DBMS C from the other experiments with JSON data.

Figure 6 presents results for the queries over binary data (SF100). MonetDB and DBMS C are faster than PostgreSQL and DBMS X

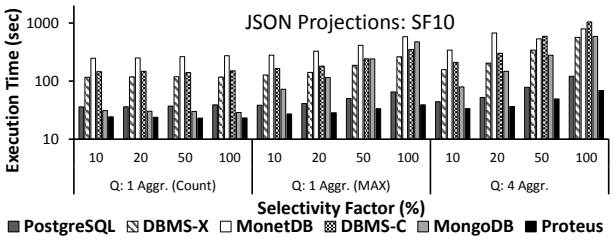

Figure 5: Projection-intensive queries over JSON data.

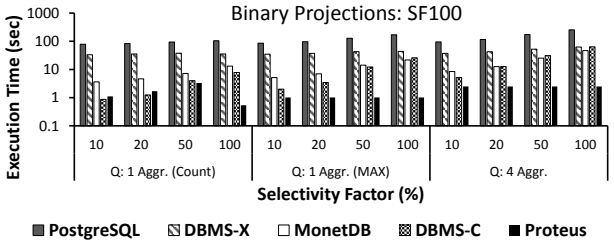

Figure 6: Projection-intensive queries over binary relational data.

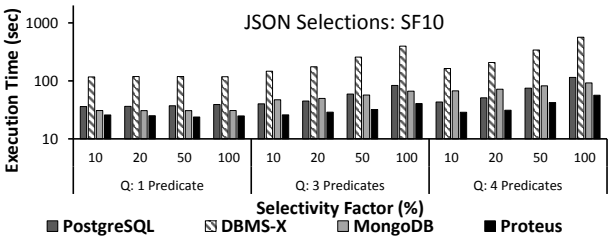

Figure 7: Selection queries over JSON data.

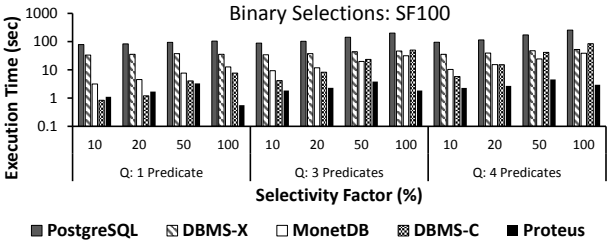

Figure 8: Selection queries over binary relational data.

because the analytical query template we study is suitable for column-oriented engines (i.e., a small subset of the relation is accessed). For selective COUNT queries, DBMS C is the fastest system because it sorts the input during data loading; given that the query has a predicate on the sorting key, DBMS C exploits it to skip many data entries while answering the query. In addition, this query does not project any attributes, therefore DBMS C does not incur any tuple reconstruction cost. For less selective instances of the COUNT query and for the other more complex queries, Proteus is faster than DBMS C and MonetDB; their columnar operators produce intermediate results (i.e., fully materialize their output), thus paying a materialization cost for the columns involved. The materialization cost increases further as queries become less selective; Proteus pipelines data instead. In addition, the resulting code of Proteus is a tight, minimal while-loop which only contains an if block evaluating the selection condition. The importance of generating minimal code is highlighted in the COUNT query (left side of Figure 6). The code is minimal enough for the effect of the branch predictor to be visible. When selectivity reaches 100%, very few mispredictions occur, therefore the query becomes faster for Proteus, although intuitively Proteus does more work to calculate the aggregate value.

Selections. To test queries with multiple selection predicates, we use three variations of the following template: `SELECT COUNT(*) FROM lineitem WHERE val1<[X] AND ... AND valN<[Z]`. The examined queries include one, three, and four predicates in the WHERE clause respectively.

Figure 7 presents the results over JSON data (SF10). Proteus has to convert the values it needs on the fly, whereas PostgreSQL and MongoDB operate over a binary serialization. Still, Proteus is faster than the other systems across the whole experiment because once it has extracted the values it needs, it reduces the rest of the CPU overheads significantly. Besides pipelining, Proteus consults its structural index to pinpoint needed fields, thus reducing navigational cost in the file. These benefits become more apparent for less selective queries. DBMS X is the slowest system because of its character-based JSON encoding. Compared to Figure 5, MongoDB closes the gap on PostgreSQL and Proteus because the current query template projects out a count instead of more complex aggregates which MongoDB does not compute as efficiently.

In the case of binary data presented in Figure 8, the outcome is similar to the one for projection queries. Proteus is faster in the majority of cases because it pipelines data through all operators. MonetDB and DBMS C operators materialize their output, which becomes more expensive as selectivity moves towards 100%.

Joins & Unnests. To test joins, we use three variations of the following template: `SELECT AGG(o.val1),...,AGG(o.valN) FROM orders o JOIN lineitem l ON (o_orderkey = l_orderkey) WHERE l_orderkey < [X]`. The first two variations compute one aggregation, COUNT and MAX respectively, while the third variation computes two aggregations (COUNT and MAX).

Document stores such as MongoDB lack first-class support for join operations, under the assumption that JSON data is typically denormalized (i.e., any joins are pre-materialized). We therefore include one more variation of a COUNT query over denormalized JSON data; each order object now contains an array with the lineitems that correspond to it, so the query has to *unnest* these JSON arrays instead of performing a join.

Figure 9 plots the results for the JSON case. Proteus is faster than the other systems because of i) its minimal generated code, ii) its lightweight JSON access path, and iii) the efficiency of the radix hash join algorithm it uses, which explains the larger performance gap from PostgreSQL compared to the previous query types. For MongoDB, we implement the join logic in a map-reduce-like query. MongoDB is unsuitable for such operations, which explains its poor performance; we only list its results for the first query as an indication. On the contrary, in the "Unnest" case, MongoDB outperforms PostgreSQL and DBMS X, which rely on built-in functions to perform data unnesting instead of an explicit query operator. Proteus is faster because its generated code involves almost no data conversions; besides evaluating a predicate, the code only increases a counter for each element of the nested *lineitem* arrays.

For joins over binary data, the query template is ideal for DBMS C and DBMS X. As seen in Figure 10, DBMS C is the fastest system for selective queries because it exploits the fact that it sorts the data on the filtering key at loading time and thus skips multiple entries. In addition, it performs sideways information passing: it applies the filter on *orderkey* to both sides of the join, thus reducing the pairs to be joined. DBMS X also performs sideways information passing, thus closing the gap with the column stores and Proteus compared to previous queries. For less selective queries, Proteus is the fastest system because DBMS X and DBMS C prune fewer tuples. To further study performance, we measure performance counter statistics for MonetDB and Proteus because they use the same query plan without the additional optimizations. For a join with 20% selec-

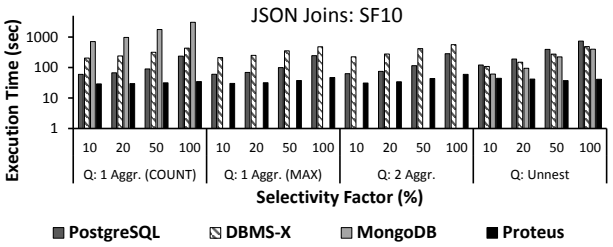

Figure 9: Join and unnest queries over JSON data.

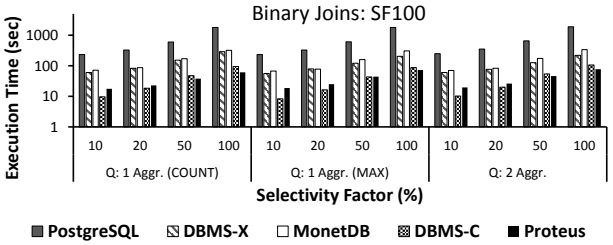

Figure 10: Join and unnest queries over binary relational data.

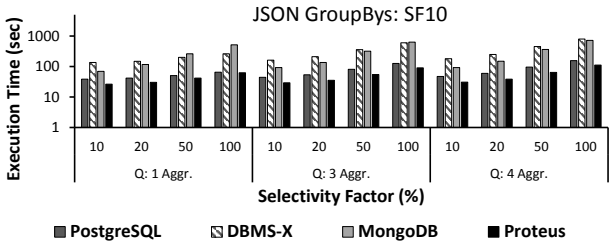

Figure 11: Aggregate queries over JSON data

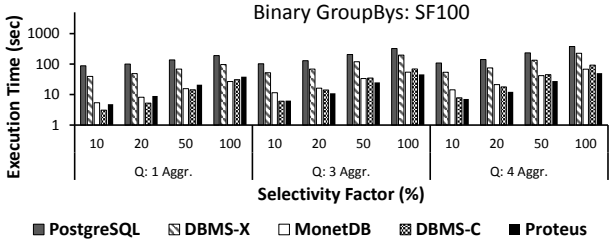

Figure 12: Aggregate queries over binary relational data

tivity, Proteus had 40× fewer dTLB (data Translation Lookaside Buffer) misses, 10× fewer last-level-cache (LLC) misses, and 2× fewer branches encountered, leading to fewer branch mispredictions. These factors contribute to faster response times for Proteus.

Aggregations. To test queries that group results, we use three variations of the following template: `SELECT AGG(val1),..., AGG(valN) FROM lineitem WHERE l_orderkey<[X] GROUP BY l_linenumber`. Figures 11 and 12 present results for queries calculating one, three, and four aggregate values. Proteus uses a radix-hash-based grouping implementation, so the results for JSON data (SF10) are similar to the join use case, with Proteus outperforming the rest. For the first query over binary data (SF100), MonetDB exploits an optimization to perform the grouping without explicitly calculating a count: It calculates the count by returning the size of each corresponding bucket in the hashtable it populates to perform the grouping. Therefore, it gradually becomes faster than Proteus when only a count is computed. DBMS C also has a head-start because it skips data based on the *orderkey* value. For queries with additional aggregates, Proteus is the fastest system.

Gauging the Effect of Caches. In the previous experiments, the caching feature of Proteus was deactivated. To quantify the speedup that Proteus can achieve by enabling caching, we instantiate the previous "projection" and "selection" templates for JSON and vary selectivity from 10% to 100%. Figure 13 plots the results. The first query applies a selection predicate and projects four fields. The "Baseline" dotted line is the Proteus configuration used in the previous experiments. In its "Cached Predicate" variation, the values used in the query's selection predicate are already cached by a previous query. The second query evaluates four predicates and then calculates a count. Its "Cached Predicate" version reads the values to evaluate the most selective predicate from the caches. In both queries, cache size is $\sim 1.2\%$ of the JSON file.

For the projection template, caching JSON values brings a high benefit. By touching the JSON file only to access the qualifying values to be projected, Proteus achieves a speed-up of up to 15× for selective queries. As selectivity reaches 100%, Proteus avoids fewer accesses of the JSON file, therefore the speedup is lower. We observe significant speedup for the selection template as well. The speedup is smaller than in the case of the projection query, because even though the projection-intensive query is more expensive than the selection-intensive one in its baseline version, both of them end up having the same execution time under "Cached Predicate". In

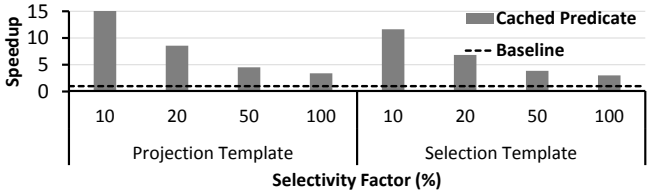

Figure 13: Effect of caching on i) a projection query and on b) a selection query over JSON data.

other words, there are some constant costs (e.g., structural index navigation) which define the minimum execution time.

Summary. Proteus is competitive with specialized systems for different operations regardless of the underlying data models and formats. We also saw the additional benefits brought by caching, which we investigate further in the next section.

7.2 Adapting to a Real-world Workload

We now evaluate Proteus using a workload obtained from Symantec, which performs analysis over data derived from spam e-mails. The data silo of Symantec periodically receives batches of JSON files, collected through worldwide-distributed spam traps. Each file contains information about spam e-mails, such as the mail body and its language, its origin (IP address, country), and the bot responsible for it. These files are the input to the data mining workflows of Symantec; classification and clustering are performed over them, through which each mail is assigned to a class per classification criterion. In every iteration of the workflow, output is stored in comma-separated-values (CSV) files containing an identifier of each e-mail, various assigned classes, etc. Finally, data is transformed and loaded in an RDBMS, with the use of which further calculations are made. This process is repeated for every new batch of JSON files: In each repetition, "fresh" JSON and CSV files have to be loaded in a DBMS and queried along with pre-existing data.

Analyzing this data involves queries over combinations of the datasets. We compare three possible solutions, for which we use i) an RDBMS that has been extended to support richer data models, ii) an RDBMS for flat data and a document store for hierarchies, and iii) Proteus, which reshapes itself based on each query. For approach I, we use PostgreSQL because it utilizes the most efficient JSON encoding out of the general-purpose systems we tested. For approach II, we use the combination of the specialized systems

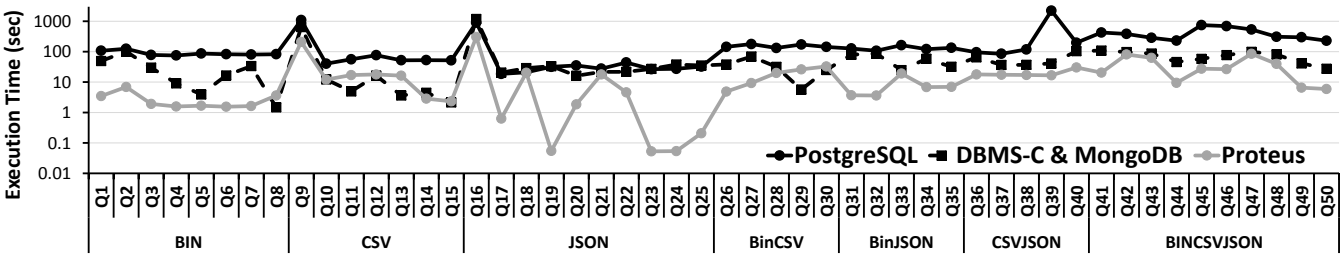

Figure 14: For a spam analysis workload, Proteus outperforms the other systems in the majority of queries due to i) its lightweight, specialized-on-demand code paths, and ii) the caches it builds as a side-effect of query execution.

DBMS C and MongoDB, along with a mediating layer on top of them to facilitate cross-format queries and data exchange.

The input comprises a 20GB JSON file of 28M objects with arbitrary field order, a 22GB CSV file of 400M records, and a 95GB database table of 500M records. PostgreSQL and MongoDB load the JSON data prior to querying it. PostgreSQL requires 22GB to store the binary JSON encoding, and MongoDB requires 30GB. Proteus builds a structural index during its first access to the JSON file; its size is $\sim 24\%$ of the file. DBMS C and PostgreSQL load the CSV data prior to querying it. Proteus again builds a structural index during the first access, storing the position of every 5th field; its size is $\sim 17\%$ of the file. Regarding binary storage, Proteus operates over binary column files similar to the ones of MonetDB. Proteus caching is enabled in this experiment. The experiment starts with the OS cache containing the binary table, and none of the systems having accessed the CSV nor the JSON data yet.

We launch a workload of 50 queries sequentially, and progressively query a variety of the datasets. The queries perform selections, 2- and 3-way joins, unnests of JSON fields, result groupings, and aggregate computations. Projectivity ranges from 1 to 9 fields, and selectivity from $\sim 1\%$ to 25%. We group together queries accessing the same datasets. We show the results in Figure 14.

Q1-Q8 touch the binary dataset. For Q1-Q7, Proteus is the fastest approach, which corroborates the findings for selection and grouping queries over TPC-H data. Q8 has a very selective predicate on the field used by DBMS C to sort the input data, therefore DBMS C skips a large part of the dataset and is slightly faster than Proteus.

Q9-15 touch the CSV dataset. For DBMS C and PostgreSQL, the execution time of Q9 includes the loading time of the CSV dataset. Proteus answers queries over the original data, also building a structural index during Q9 and caching any fields it converts to answer the query. Q9 takes Proteus 440 seconds less than DBMS C, and 880 seconds less than PostgreSQL. DBMS C is faster than Proteus in Q11 because it operates over binary data, whereas Proteus converts data fields on-the-fly and pays to cache them for further use. Indeed, Proteus partially serves Q12 from its caches. On the other hand, Q12 also has a filtering predicate on a string field. Proteus opts not to cache string fields, whereas DBMS C performs dictionary encoding of string values during loading and exploits it in Q12; still, both systems have similar performance. Q13 is also heavy on string-based operations, which explains why DBMS C is faster. For Q14 and Q15, Proteus is the fastest approach because of the binary caches it populates and its minimal generated engine.

Q16-Q25 touch the JSON dataset, so MongoDB becomes active. For Q16, all systems behave as in Q9: Proteus exploits that the JSON dataset is accessed for the first time, and caches data aggressively since the caching cost is masked by I/O and the structural index construction. Q16 takes Proteus 600 seconds less than MongoDB, and 800 seconds less than PostgreSQL. For Q17, Proteus uses its caches to speed-up execution significantly. For Q18 and Q21, caches are less useful because the queries involve string fields which Proteus extracts and processes from the JSON file at query time. Using a policy of caching strings would benefit Proteus in the short term, but it would also pollute the caches with string objects. Still, Proteus is slightly faster than the other systems. For the rest of the queries, custom code generation combined with judicious data conversions and adaptive caching make Proteus faster.

	Load CSV	Load JSON	Middle-ware	Q39	Queries (Rest)	Total
PostgreSQL	1019	792	0	2226	7468	11505
DBMS-C &MongoDB	711	1067	43	29	1810	3660
Proteus	0	0	0	17	1231	1248

Table 3: Execution time per Symantec workload phase.

Q26-Q30 join binary and CSV data. The materialization overhead of DBMS C is insignificant because these queries are very selective. Still, Proteus is faster for Q26 and Q27. Likewise, it is barely noticeable for Proteus that Q28 includes predicates on string fields of the CSV file. DBMS C is faster for Q29 because Proteus again has to access a string field in the CSV file, and at the same time DBMS C skips multiple data entries because of a filtering predicate on its sorting key. In general, both DBMS C and Proteus offer competitive performance for this query range. Finally, Q31-Q35 join binary and JSON data, Q36-Q40 join CSV and JSON, and Q41-Q50 join all three datasets. Q39 is very expensive for PostgreSQL because it picks a sub-optimal, nested-loop-based plan. Proteus is consistently the fastest system for two reasons: First, as discussed in Section 7.1, customizing the query engine gives significant performance benefits. Second, Proteus adaptively caches accessed values, thus after some point it largely operates over its binary caches, instead of the verbose CSV and JSON datasets.

At the end of the workload, the cache size for the CSV data is $\sim 30\%$ of the CSV file. The cache size for the JSON data is only $\sim 2.5\%$ of the JSON file. JSON caches are more compact because although the number of CSV and JSON fields of interest is almost the same, the JSON file contains 28 million verbose JSON objects to be partially cached, whereas the CSV file contains 400 million narrow tuples. Interestingly, the JSON caches are more impactful for the workload because of the increased access cost for the JSON dataset. Therefore, if we were to drop any caches to adhere to a tighter memory budget, we would start from the ones for CSV data.

Aggregate Performance. Table 3 presents the accumulated execution time spent in each workload step. Proteus is $9.12\times$ faster than using an RDBMS with added support for richer data models (PostgreSQL) and $2.9\times$ faster than the approach of packaging together multiple query engines and using the appropriate one for each specialized scenario (DBMS C & MongoDB). We isolate Q39 because it is an outlier for PostgreSQL that highlights the problem of extending existing systems without deeply integrating support for the added data models and formats. Q39 performs a join between the CSV and JSON datasets. PostgreSQL, however, treats

JSON data as a BLOB-like datatype, which is essentially opaque to its optimizer. The result is that the optimizer chooses an expensive nested-loop join. If we exclude Q39 from the aggregated execution time, Proteus is still 7.4× faster than PostgreSQL. Finally, even if we focus completely on execution time and exclude any other overheads from the workflow (e.g, data loading cost, overhead of middleware layer), Proteus still is the fastest system overall.

Summary. Proteus flexibly accesses a real-life workload of heterogeneous datasets while being as fast as a specialized system per use case. Besides being fast regardless of its input, Proteus achieves an additional speed-up by adapting to the workload through caching structures built as a side-effect of querying.

8. CONCLUSION

Data analysis solutions over heterogeneous data have always involved a trade-off: be flexible and serve diverse datasets at the cost of performance, or be rigid and specialized for a specific scenario, thus leading users to employ a different system per use case.

This paper presents a system design that exposes heterogeneous datasets under a single interface, while exhibiting the response times of a system specialized per use case. The design couples i) an expressive query algebra which masks data heterogeneity with ii) on-demand customization mechanisms which produce a new system implementation per query. Based on this design, we build Proteus, a query engine that natively supports CSV, JSON, and relational binary data, and specializes its entire architecture to each query and the data that it touches via code generation. Proteus also customizes its caching component, specifying at query time how these caches should be shaped to better fit the overall workload.

Proteus serves efficiently synthetic and real-world workloads: it outperforms state-of-the-art open-source and commercial approaches without being tied to a single data model or format, all while operating transparently across heterogeneous data. Its ability to morph per query opens multiple opportunities for further optimizations.

Acknowledgments. We would like to thank the reviewers for their valuable comments and suggestions on how to improve the paper. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa).

9. REFERENCES

[1] Apache Drill. https://drill.apache.org/.
[2] LLVM's Analysis and Transform Passes. http://llvm.org/docs/Passes.html.
[3] A. Abouzeid et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.
[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.
[5] I. Alagiannis et al. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.
[6] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: a hands-free adaptive store. In *SIGMOD*, 2014.
[7] S. Alsubaiee et al. AsterixDB: A Scalable, Open Source BDMS. *PVLDB*, 7(14):1905–1916, 2014.
[8] M. Armbrust et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.
[9] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.
[10] R. Barber et al. Business Analytics in (a) Blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.
[11] K. S. Beyer et al. System RX: one part relational, one part XML. In *SIGMOD*, 2005.
[12] K. S. Beyer et al. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.
[13] S. Blanas et al. Parallel data analysis directly on scientific file formats. In *SIGMOD*, 2014.
[14] P. Boncz et al. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, 2006.
[15] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.
[16] R. Brunel et al. Supporting hierarchical data in SAP HANA. In *ICDE*, 2015.
[17] F. Bugiotti et al. Invisible Glue: Scalable Self-Tuning Multi-Stores. In *CIDR*, 2015.
[18] M. J. Carey et al. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *RIDE-DOM*, 1995.
[19] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In *WebDB*, 2013.
[20] S. S. Chawathe et al. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*, 1994.
[21] Y. Cheng and F. Rusu. Parallel In-Situ Data Processing with Speculative Loading. In *SIGMOD*, 2014.
[22] D. J. DeWitt et al. Split Query Processing in Polybase. In *SIGMOD*, 2013.
[23] J. Duggan et al. The BigDAWG Polystore System. *SIGMOD Record*, 44(2):11–16, 2015.
[24] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.
[25] M. F. Fernández, J. Siméon, and P. Wadler. An Algebra for XML Query. In *FST TCS*, 2000.
[26] S. J. Finkelstein. Common Subexpression Analysis in Database Applications. In *SIGMOD*, 1982.
[27] G. Graefe and W. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *ICDE*, 1993.
[28] M. Grund et al. HYRISE - A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.
[29] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
[30] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.
[31] M. Ivanova, M. Kersten, N. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, 2009.
[32] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.
[33] M. Karpathiotakis et al. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*, 2015.
[34] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.
[35] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In *SIGMOD*, 1999.
[36] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.
[37] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.
[38] Z. H. Liu, B. C. Hammerschmidt, and D. McMahon. JSON data management: supporting schema-less development in RDBMS. In *SIGMOD*, 2014.
[39] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing main-memory join on modern hardware. *IEEE TKDE*, 14(4):709–730, 2002.
[40] S. Melnik et al. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.
[41] R. Murthy et al. Towards an enterprise XML architecture. In *SIGMOD*, 2005.
[42] F. Nagel, P. A. Boncz, and S. Viglas. Recycling in pipelined query evaluation. In *ICDE*, 2013.
[43] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.
[44] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
[45] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *ICDE*, 2006.
[46] M. T. Roth, F. Ozcan, and L. M. Haas. Cost Models DO Matter: Providing Cost Information for Diverse Data Sources in a Federated System. In *VLDB*, 1999.
[47] M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB*, 1997.
[48] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, 2000.
[49] A. Shaikhha et al. How to Architect a Query Compiler. In *SIGMOD*, 2016.
[50] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.
[51] M. Stonebraker. Technical perspective - One size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.
[52] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: a SQL system for multi-structured data. In *SIGMOD*, 2014.
[53] A. Thusoo et al. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.
[54] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE TKDE*, 10(5):808–823, 1998.
[55] P. W. Trinder. Comprehensions, a Query Notation for DBPLs. In *Database Programming Languages: Bulk Types and Persistent Data.*, 1991.