Boosting Efficiency of External Pipelines by Blurring Application Boundaries

Anna Herlihy
anna.herlihy@epfl.ch
EPFL
Switzerland

Periklis Chrysogelos
periklis.chrysogelos@epfl.ch
EPFL
Switzerland

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL, RAW Labs SA
Switzerland

Abstract

Modern application development addresses increasingly specialized problems using domain-specific utilities, such as Optical Code Recognition and standalone statistical tools. The diversity of tooling, combined with the ever-growing volume of data, requires data pipelines to be both efficient and support a variety of data processing tools within the same pipeline. Existing approaches, however, impose a tradeoff between modularity and performance: on the one hand, data processing systems are specialized for fast execution of complex queries, favoring efficiency at the expense of high development costs and required domain expertise. On the other hand, highly extensible systems opt for composability at the expense of inefficient execution due to minimal assumptions about input and output formats.

This paper proposes Generalized OLAP (GOLAP), a new DBMS paradigm that places automatic extensibility of functionality as a first-class design goal. GOLAP ingests external utilities to achieve the functionality provided by external modular data pipelines while maintaining the performance of natively optimized DBMS functions. Through a combination of runtime inspection and static analysis, GOLAP detects inter-utility communication inefficiencies and parallelization opportunities beyond the limits of isolated utility optimizations. It then modifies the utilities to elide unnecessary inter-utility operations and parallelizes the pipeline to increase hardware utilization. To evaluate GOLAP, we build Caesar, a prototype that optimizes simple pipelines, showing up to 22x speedup while introducing a limited instrumentation period with a slowdown of less than 17%.

1 Introduction

The ever-growing demand for sophisticated applications creates an unprecedented need for complex, domain-specific, and heterogeneous building blocks that surpass the capabilities of traditional data processing engines. Fortunately, data practitioners have a plethora of ready-made building blocks at their disposal, ranging from new statistical tools to specialized utilities like Optical Code Recognition (OCR) to extract text from images to system monitoring tools. However, the absence of a do-it-all infrastructure creates a dilemma: sacrifice performance and create multi-system pipelines, or wait for all the required utilities to be implemented on the same framework.

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

While integration into a unified framework allows global optimizations as well as a consistent type system and data representation, it requires time-consuming manual effort. Thus it is often limited to popular utilities, like the latest machine learning algorithms. Nevertheless, the fast pace of application development and the wide range of developer expertise leads to the reuse of libraries created for a single purpose. As a result, data practitioners prioritize components that serve their purpose – even at the expense of interoperability overheads such as utilities communicating via a human-readable, e.g., CSV/JSON, intermediary format.

To minimize the interoperability cost, existing works follow two main directions: building adaptive systems or transcompiling existing utilities. The first category relies on having modular system designs that virtualize data accesses [14, 15] to reduce the cost of operating on raw data, by requiring all utilities to adhere to a custom system-specific architecture. The second category relies on software lifting techniques [5, 6, 9] to lift a lower-level, usually imperative, program description to a higher-level optimizable representation. This comes with the requirement that the program is representable in the higher-level Domain Specific Language (DSL) – thus restricting the utilities that can be lifted to ones that the system can understand. In both cases, the focus is on optimizing the utilities internally.

Nevertheless, many common overheads originate from inefficiencies due to the integration of utilities and not due to inefficient implementation of their internal core logic: gluing together multiple small, off-the-shelf utilities introduces costly materializations beyond the responsibility bounds of any single utility. Similarly, exploiting the available hardware parallelism requires orchestrating the utilities to exploit data and task parallelism – often beyond the implemented intra-utility parallelization. As a result, existing approaches create a tradeoff between expressiveness, i.e., using off-the-shelf coarse-grained components to express the required operations, and performance, i.e., having an efficient runtime.

This work presents GOLAP – an ecosystem-centric architecture that allows engines to extend their functionality without sacrificing performance by inspecting and optimizing data pipelines. GOLAP optimizes interoperability and orchestration of data pipelines by i) using both compile and runtime information to detect and remove unnecessary serialization operations, ii) using orchestration contracts to parallelize and batch data pipeline operations. In contrast with previous approaches, GOLAP improves interoperability overheads by optimizing utilities near their boundaries instead of trying to fully understand the core functionality of each utility. To evaluate the applicability of GOLAP, we build Caesar, a preliminary prototype that optimizes data pipelines to achieve up to 22x speedup in end-to-end execution latency compared to the unoptimized

```
base="https://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year";
for y in {2010..2020}; do
  curl -s "$base/$y.csv.gz" |
  gunzip | noaa-decoder | range-calc |
  sed -e "s/ *$//" | xargs -I{} grep {} ghcnd-stations.txt |
  sed "s/^/Station with the largest temperature range in $y:"
done
```

Figure 1: Example analytical pipeline with a mix of utilities

data pipeline and similar performance to manually optimized ones. Overall, GOLAP enables a new breed of analytical systems that auto-expand their functionality by absorbing external utilities.
This paper makes the following **contributions**:

- We identify that off-the-shelf components provide fast access to new functionality while also providing an initial implementation that is amenable to optimizations and automatic tuning.
- We provide GOLAP, a new DBMS design that absorbs functionality by inspecting external pipelines to avoid interoperability overheads and missed parallelization opportunities – a significant benefit for DBMS-native operations.
- Through Caesar, we evaluate GOLAP on a mix of analytical and OCR-enabled pipelines and show up to 22x speed-up in end-to-end execution latency.

2 Motivating Example

Modern data pipelines target complex data collection and processing tasks. For example, a data pipeline may fetch images of handwritten notes, apply an Optical Code Recognition (OCR) method to extract the text, and run analytics on the result. Pipelines may be even more domain-specific, for example translating proofs from scans of old textbooks into an automatic theorem prover to create a database of theorems. While these are two very specific examples, they illustrate that the diversity of modern applications 1) requires data pipelines to be easily expressed in high-level, often domain-specific terms ("verbs") and 2) has made specialized examples the norm rather than the exception.

While database vendors put in significant effort to incorporate new functionality, support lags for less popular utilities. This impacts expressiveness, as data practitioners cannot introduce custom verbs into pipelines without sacrificing performance. If the database engine does not support functionality like fetching data from the network based on a subquery, uncompressing images, or running OCR, then the user is forced to invoke UDFs or external pipelines and tools. However, the effectiveness of the database performance optimizations reduces as the main workload shifts towards the chained external utilities.

As an alternative, we make the case for first-class support for expressing data pipelines using ready-made applications. Traditionally, there have been two extremes: on one side, analytical engines optimize for high performance of data pipelines expressed in the engine's native (e.g., relational) algebra. On the other side, bash/Python/serverless-like gluing frameworks optimize for expressiveness and composability at the expense of interoperability overhead. In this work, we show that optimizing interactions between applications allows data practitioners to express complex tasks in high-level terms without sacrificing performance.

Figure 1 depicts a modified version of a data pipeline from PaSh [30] that fetches data from the National Oceanic and Atmospheric Administration (NOAA) to find which weather monitoring station had the highest single-day temperature range per year. We use this pipeline as an example of a traditional analytical query, one that uses a set of off-the-shelf general utilities (`curl`, `gunzip`, and `sed` to fetch, decompress and post-process the data, respectively) as well as two task-specific utilities (`noaa-decoder` processes data to extract relevant values and `range-calc` groups the input CSV by the first column and simultaneously computes the max of the second column).

The NOAA pipeline is put together using `bash` to pipe data from one utility to the next. For the rest of the text we use bash-like terminology to describe the pipelines and application interactions; however GOLAP is not a bash optimizer. GOLAP is a proof-of-concept for a DBMS design that ingests analytical pipelines: we consider data pipelines composed of applications (utilities) that communicate through explicit pipes. At the moment, the individual operators are executables that read data from stdin and write it to stdout. Our implementation and examples use C++ for the application code; however, our proposal can be extended to ingest and optimize other source languages. Furthermore, for utilities to take advantage of optimizations that are based in just-in-time inspections and source code modifications, we assume the source code and build system of the pipeline's applications is available.

While the NOAA pipeline requires minimal development effort, it misses standard OLAP performance optimizations such as parallelization and suffers from unnecessary serializations. Expressing the same pipeline in an OLAP engine, however, requires operations that are often missing from a DBMS, such as task-specific functionality like `noaa-decode`. *In this paper, we propose GOLAP: a system design that targets both expansion of DBMS functionality and efficient execution of external utilities, through inspection and rewriting of data pipelines.*

3 Background and Related Work

Optimizing data pipelines has received attention from both the database and compiler communities. This section surveys the related work and explains how GOLAP extends the state-of-the-art.

Efficient analytical query execution. Analytical engines have powered pipelines and heavy-duty warehousing queries for decades. OLAP engines use internal properties to parallelize execution. Specifically, they inject meta-operators [10] or use parallelized operator variants [20] to achieve efficient, scalable execution. Vectorization [2], code generation [22], and their combination [17, 21] have ameliorated indirection and inter-operator overheads. Furthermore, just-in-time code generation allows analytical engines to execute queries over various data formats [14, 15] efficiently. As external utilities are outside GOLAP's control, it uses orchestration contracts and utility rewrites to achieve similar optimizations.

Auto-managed pipelines. The complexity of modern workloads has led to the transition to automanaged, serverless environments that provide orchestration as a service, inspiring work on optimizing execution in the cloud [6, 13] and on scalable operating systems [3] to support it. As a core component of these infrastructures, including Docker, the UNIX Shell is at the frontend [11, 25], with PaSh [30] parallelizing scripts through annotations. Lastly,

speculation [26] and auto-parallelization through look-ahead predictions [18, 31] expose additional parallelism. GOLAP uses orchestration artifacts similarly to PaSh but is not Shell specific and modifies the application source to further optimize execution. GOLAP provides a runtime layer for running applications similarly to DBOS [3], but instead of providing efficient OS primitives, it modifies the execution layer itself.

Domain-Specific Languages (DSLs) allow expressive and optimizable representations for many domains, including for data-intensive [23] and machine learning [4] tasks. Furthermore, verified lifting [5, 6, 9] allows the lifting of low-level representations into higher ones if the higher-level DSL supports the low-level operations. However, finding a DSL that has broad applicability is an open challenge. Thus, MLIR [19] proposes loosely combining multiple DSLs through a generic representation that allows co-optimizing multiple representation levels. GOLAP follows a multi-representation strategy, similarly to MLIR, and lifts input programs, similarly to verified lifting. In contrast to both MLIR and verified lifting, GOLAP uses runtime information to lift the input programs and optimize them.

Code optimization through traces. Profile-Guided Optimization (PGO) [24] uses traces to guide compiler optimizations. Partial Evaluation (PE) [27, 32] modifies the binary just-in-time, to aggressively inline and improve branch handling inside the boundaries of an single utility/application. In this work we build on top of PE to 1) enable inter-utility optimizations, and 2) take advantage of the limited operations on intermediary (cross-application) data to extend these techniques to modify the intermediary data representation. Similarly to JVM JIT compilers [32], GOLAP does not apply optimizations if assumptions are invalidated during runtime. Weldr [12] fuses binaries to allow inter-application analysis. Both PGO & Weldr operate at compile time and thus have to generate programs equivalent to input ones – prohibiting aggressive optimizations such as changing output formats that GOLAP can perform by modifying utilities just-in-time. Furthermore, in contrast with PE, GOLAP has multi-application visibility and thus gathers and synchronizes information across utilities.

4 Where Time Goes

Despite the plethora of performance optimizations described above, connecting multiple off-the-shelf utilities into data pipelines still suffers from inefficiencies: application boundaries hinder inter-process optimizations and orchestration, resulting in a trade-off between modularity and performance. This section discusses these two sources of inefficiency and how the increasing diversity of tooling challenges existing database solutions.

4.1 Interoperability & Intermediate Data

What happens. Big data-intensive applications and frameworks often support multiple input/output formats, from human-readable ones like CSV to high-performance, standardized ones like Protocol Buffers. Same-process pipelines may also rely on language-native containers such as NumPy arrays, or `C++ std::vectors`.

The problem. Pipelining from a data-producing utility to a data-consuming utility requires both applications to support at least one common (physical) data format, potentially through a third (converter) utility, as well as compatible input shapes (logical layout) like requiring that no extra columns are outputted by a producer to a consumer that can not handle them.

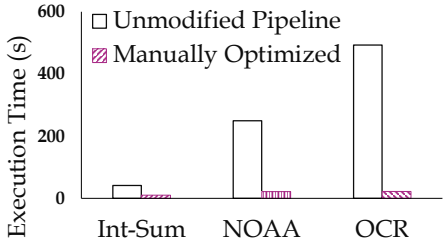


Figure 2: Potential speed-up by inter-process optimizations

The traditional way. In traditional DBMS architectures, operators are composable by design: the engine can connect logical operators to create complex execution plans. The valid plans satisfy the requirement for compatible logical layouts across operators. For the physical layout, multiple alternatives have been proposed, with either a fixed set of data representations across the operators, characterized mostly by volcano-based [10] and vectorized execution engines [1, 2], or by more flexible intermediate representations across fused operators [22] and just-in-time data access paths that span multiple operators [14, 15]. In all cases, as the data exchange across operators resides on the critical path, existing DBMS architectures either make strong assumptions about the data types and compatibility across operators or invasively require the operators to be expressed using the DBMS framework internals.

Limitations. End-to-end pipelines that rely on off-the-shelf components can not make assumptions about what frameworks each component uses, nor expect components always to produce the optimal output shape. Instead, pipelines often rely on human-friendly formats as a default (for when components are physically incompatible) or intermediate transformations (for when they are logically incompatible). Nevertheless, performance-critical data conversions often apply the same operations repeatedly, providing the potential to either avoid conversion costs by fusing applications (e.g., making calls through the internal APIs) or by injecting alternative input paths that impose lower overheads.

Example. Figure 2 shows the execution time for three pipelines, two of which have unnecessary materializations. The first one, "Int-Sum", is an integer generator (i.e. a producer utility) that pushes data into a summation aggregation (i.e. a consumer utility), while the second one is NOAA of Section 2 which also follows the producer-consumer pattern. The aggregation stages of both the Int-Sum and NOAA convert the output elements back to integers to perform calculations, which is an avoidable overhead. Knowing both sides of the pipe, a rewrite of the Int-Sum would provide up to 4.5x speed-up. In the NOAA pipeline, this optimization can be combined with orchestration optimizations described in the next section.

4.2 Orchestration

What happens. Popular applications often support parallel execution, either automatically or at the user's request. Smaller utilities may not be parallelized, or they may observe small performance

gains from multiple threads. Non-parallelized utilities may further be categorized in i) utilities that support running multiple of their instances concurrently, usually relying on process-level isolation and task parallelism, or ii) strictly sequential implementations that modify common state (e.g. files) in a multi-task unsafe manner, leaving little room for multi-core execution.

The problem. Unfortunately there is minimal support for inter-utility parallelization: utilities are developed in silos, leaving little space for automatically parallelizing chains of utilities. Existing approaches rely on the user manually spawning multiple pipelines, with the exemplary cases of work queues and serverless infrastructures for parallelizing independent tasks. As a result, utilities with sufficient data parallelism can self-parallelize in isolation but parallelizing combined pipeline stages is left to the user due to the architectural separation imposed across utilities for modularity.

The traditional way. Traditional DBMS designs embrace task- and data-parallelism even in the presence of strict, e.g. ACID, isolation requirements. For data-parallelism, big tasks are broken into multiple small ones by parallel OLAP engines to exploit multiple accelerators and CPUs. This is accomplished by leveraging different operator properties in two ways. First, each execution stream is read-only and the mutable state is private [7], and the execution plan handles stream and intermediate data structure consolidation. Second, the operators themselves handle inter-stream conflicts [8, 20]. Task-parallelism is exploited during multi-query execution or by transactions that rely on concurrency control to handle conflicts. In all cases, existing DBMS designs rely on cooperative concurrent execution across different DBMS components and knowledge about the functional characteristics of the queries.

Limitations. The limited number of database operators makes transcribing the required properties to optimization rules feasible; however, the diversity of utilities hinder a similar approach because of sheer volume. Parallelization opportunities are further decreased due to the architectural boundary that limits the exposure of parallelism by applications themselves: a producer utility can not control the parallelization level of the utility consuming its output. Instead, existing pipelines rely on the user explicitly orchestrating the utilities to achieve efficient hardware utilization.

Example. The default NOAA and OCR pipelines (Figure 2) suffer from hardware underutilization as parallelization is only within a utility. Parallelizing the OCR pipeline leads to a 22x speedup over the default execution, while parallelizing and avoiding the serialization results in 17x speedup for the NOAA pipeline.

5 Generalized OLAP: Blurring the Boundaries

We envision GOLAP, a DBMS architecture that is ecosystem-aware: the DBMS understands and ingests applications to optimize the overall execution and enable efficient, generic data pipelines. GOLAP embraces composability by allowing applications to form data pipelines while at the same time applying optimizations by discovering pipeline characteristics and restructuring both the pipeline execution as well as the applications themselves. While traditional database architectures only optimize internal execution, GOLAP i) makes external utilities a first-class execution primitive, ii) learns from pipeline descriptions and optimizes inter-utility operations.

Diverse ecosystem as an opportunity. Instead of trying to mimic and re-implement the continuously diversifying requirements of data pipelines, GOLAP embraces external utilities and uses them as a means of adopting new functionality. GOLAP is based on the observation that the pipeline description (i.e. the ordered list of executables and their arguments) is itself a starting point for the DBMS to learn functionality, by inspecting and executing the provided pipeline. For example, each stage of the NOAA pipeline example in Figure 1 would be an element in the ordered list that makes up a pipeline description. GOLAP can then focus on finding and optimizing inefficient stages by learning from the description and inspecting data that flows though the pipeline.

Domain experts write the core logic of utilities. However, the interfaces used to connect and interact with other utilities often lack optimizations – application interfaces are designed based on expected, not actual, usage patterns . GOLAP inspects the pipeline statically and dynamically to discover and fix inefficiencies on the utility boundaries, and improves invocation patterns that are otherwise outside the responsibility of any single component.

Understand inefficiencies – not operations. Ingesting external utilities removes the burden from the DBMS to understand what logical operations are being performed, as long as it has access to and can transform the utility source to bypass inefficiencies. Furthermore, while general optimization of each utility is challenging, the GOLAP architecture simplifies the task by reducing the problem to identifying specific optimizable patterns. Optimizing pipelines just-in-time allows GOLAP to collect runtime data about the utilities and their invocation parameters. This data informs decisions about optimization and de-optimization.

Blueprint. An orchestration environment should represent the pipeline in a form that is amenable to optimization. Creating a Domain Specific Language (DSL) that encapsulates all the required traits creates a trade-off between the complexity of the DSL and the information it encapsulates. Each optimization, however, requires only a subset of the traits and thus can be best served by a DSL that is made-to-fit the corresponding optimization.

GOLAP combines multiple optimizations by allowing the system to operate over each optimization's preferred representations, as long as there is a one-to-one mapping back to full utilities. That is, optimizations are applied by GOLAP using the optimization-specific representation which must include a reference back to the original utility. Then, the system can inject instrumentation to inspect the applications on their runtime and perform rewrites.

Benefits of GOLAP. First, external utilities benefit from the internal DBMS performance optimizations, while the DBMS benefits from externally implemented functionality without the overheads and limitations of UDFs. Second, by optimizing utilities based on real-time ecosystem and workload information, GOLAP allows informed optimization decisions based on how the utility is used. Instead of optimizing applications a priori and relying on expected workloads, GOLAP gathers the optimization and usage information from the execution environment. Thus expensive yet required functionality for modularity, like managing a multitude of supported formats, can be omitted. Third, GOLAP avoids transforming the full application semantics, allowing utilities to be ingested and optimized by lifting only inefficient operations that take place near the inter-utility boundary.

Challenges. Materializing GOLAP relies on three pillars.
Challenge #1: representation. GOLAP engines need to optimize a diverse range of applications with unknown semantics. For GOLAP to apply the different optimizations, it needs a DSL that is amenable to multiple transformations but simple enough for a GOLAP implementation to lift the data pipelines into its DSL.
Challenge #2: understanding. GOLAP relies on compile and runtime inspection to understand parts of utilities. Verified software lifting [5, 6] frameworks aim to understand the core functionality of utilities to transform them to an equivalent IR natively supported by the framework. Lifting, however, requires the principle concepts of the core functionality to be compatible with the target framework and simple enough for the transformation to succeed. The GOLAP architecture reduces the concepts that must be learned by targeting hardware underutilization & interoperability inefficiencies across utilities. This creates a new set of challenges for software lifting and an opportunity for aggressive, focused, partial lifting, but requires GOLAP to understand the application boundaries and identify inefficiencies such as inverse operations.
Challenge #3: modification. GOLAP relies on transforming pipelines to avoid overheads on multiple levels. Nevertheless, avoiding overheads like unnecessary serialization requires modifying the utilities. Thus, GOLAP engines need to rewrite utilities not designed for code generation and structure modifications. Many code generation frameworks rely on either getting a source language and lowering it into a lower-level one or into code-generation-aware infrastructures where the components actively tune themselves based on their inputs to generate specialized versions. Nevertheless, GOLAP follows a learn-by-example paradigm where it starts with the unmodified utilities, and despite the utilities being oblivious to the code generation, it modifies their source code. As a result, GOLAP engines need to modify existing applications without breaking their semantics in the scope of the current pipeline.

6 Caesar: A GOLAP Orchestration Engine

To evaluate the benefits and the feasibility of GOLAP, we built Caesar, a prototype orchestration engine that instruments and optimizes simple pipelines. To this end, Caesar implements simple versions of the main components required to optimize example pipelines, leaving generalization and equivalence guarantees needed by the end-to-end pipeline for future work. Caesar accepts a pipeline description, as well as optional testimonials for operators that are used to apply orchestration and serialization optimizations. The testimonials include the classification of utilities for parallelization and the location of the source code for serialization optimizations.
Caesar overview. To cover multiple cases across the optimization spectrum described in Section 4, Caesar provides prototype support for high-level pipeline transformations, i.e., process-level parallelization and low-level pipeline transformations, i.e., serialization avoidance. To apply low-level pipeline transformations, Caesar follows a three-phase cycle corresponding to the challenges outlined in Section 5: i) it creates an optimization-specific representation of the pipeline, ii) it inspects the description as well as the runtime behavior, and iii) optimizes the pipeline by transforming the pipeline, the utilities that compose it, or both.

Veni: pipeline representations. During optimization, Caesar keeps two levels of representation: one in terms of the whole, executable-level utilities, and one as a parsed version of the utilities' source code. If the source code is unavailable, Caesar can still run and parallelize the utilities but cannot make low-level modifications. The executable-level representation is always maintained and is used to execute the pipeline by connecting and controlling the binaries. The source-language representation of each utility is used for low-level transformations and for injecting the data instrumentation.
Representations for parallelization. The parallelization transformation operates directly on the executable-level representation, as rewrites only change the invocation pattern of utilities. For example, a pipeline that contains a series of embarrassingly parallel utilities in a row would be transformed by greedily combining the representations of the utilities until a utility that breaks the pipeline is reached. The combined utilities would be passed to a gnu-parallel [28]-like utility which would replace the series of individual utilities in the optimized pipeline. Thus, both the final and the intermediate pipeline representation are in the executable-level representation. Specifically, the representation is a DAG, with each node representing an application and its command-line arguments connected based on the communication patterns. Caesar currently supports only explicit pipes for inter-process communication.
Representations to avoid serialization. In contrast to parallelization, avoiding serialization requires fine-grained information about the conversions performed by each utility to write its output and read its input. Thus, for serialization avoidance, Caesar inspects the source code of each utility to find the operations related to the output and input streams. Avoiding a serialization step requires finding inverse operations, such as `std::to_string` and `atoi`, across the utilities and rewriting the source code to transfer the data in its original, binary form, instead of the serialized values. The modified source code is compiled, and the representation is then lifted back into executables to maintain compatibility with the rest of the optimizations. Furthermore, unlike traditional lifting techniques, the core, non-serializing code is left untouched, so Caesar avoids the overhead of analyzing the entire utility.
Vidi: pipeline inspection. To transform the executing pipelines, we use orchestration contracts (testimonials) to parallelize the pipeline: by inspecting the pipeline, Caesar recognizes and matches utilities to their (developer-provided) testimonials in order to parallelize subpipelines. We also inspect the execution graph to find hot paths both in terms of frequently invoked utilities as well as frequently invoked utility-internal functions. Then, if there is a path that is sufficiently hot to optimize, we inject logging operations to find inverse operations across utilities.
Parallelization through testimonials. We rely on *testimonials* that categorize the utilities into parallelization categories to increase hardware utilization. Furthermore, testimonials describe how the utility should be transformed to enable parallel execution for each category. Specifically, testimonials assign each utility to a relational operator category and provide modification rules for the utility parameters that enable parallelization when possible. Testimonials can categorize utilities as projection/filter-, aggregation- and join–like, to allow optimization such as parallelizing the corresponding relational operators by moving a parallelization point

before the corresponding operation. Currently, Caesar supports only projection-like testimonials and uses them to parallelize multiple invocations of the same utility.

In contrast to manual parallelization of a pipeline, testimonials are utility-specific but pipeline-independent. As a result, testimonials are shared across pipelines and inherited during utility rewrites (e.g., after removing serialization). Furthermore, as testimonials provide only a limited number of parallelization categories, they also create structured information for the parallelization passes to create long parallel pipelines. While we plan, in future work, to add support for adaptively deciding based on runtime information the parallelization degree of each segment, for now, Caesar parallelizes only projection-like testimonials and it greedily combines the ones corresponding to a pipeline's utilities to parallelize execution.

Runtime instrumentation to bypass serialization. To avoid serialization and deserialization, Caesar first statically searches the utility source code for pairs of points consisting of a point in the producer utility that writes serialized data into the pipe, and a point in the consumer utility that reads the data and deserializes them. When such a pair is detected, Caesar has to build enough confidence that these points perform inverse operations before proceeding to the rewrite phase. While logic-based static analyses like points-to or dataflow analysis can provide strong guarantees if all the information required is available at compile-time, they are pessimistic, often computationally expensive and may fail to provide specific enough guarantees when the required information is only available at runtime. Because of these limitations, Caesar currently uses i) light source-code analysis based on AST pattern matching to identify candidate points, and ii) code instrumentation to build the required confidence by injecting code to log and compare the inputs and outputs of the candidate points. As a result, the tunable confidence allows Caesar to perform more aggressive optimizations and with less analysis effort when inaccuracies are tolerable. Caesar removes the logging if inspecting the logs proves that the two points are not inverse. In contrast, if the logs collect enough proof that the two points perform inverse operations, Caesar proceeds into the rewrite phase to bypass the serialization. To show the potential of the aforementioned process, Caesar detects `std::cout.write` of integer-to-string conversion patterns and their inverse; we leave further extending point discovery to future work.

Vici: pipeline rewrite. Based on the pipeline representations and inspection, Caesar rewrites the pipelines to take advantage of optimization opportunities in two levels: coarse-grained parallelization and fine-grained serialization avoidance.

Rewrites for parallelization. The parallelization testimonials allow for task-parallelism, by spawning multiple instances of the same utility for different inputs, such as the tasks generated by the for-loop of the NOAA pipeline. In order for Caesar to handle such transformations, it handles the pipeline execution by launching and connecting the utilities itself instead of relying on a shell or another runtime environment.

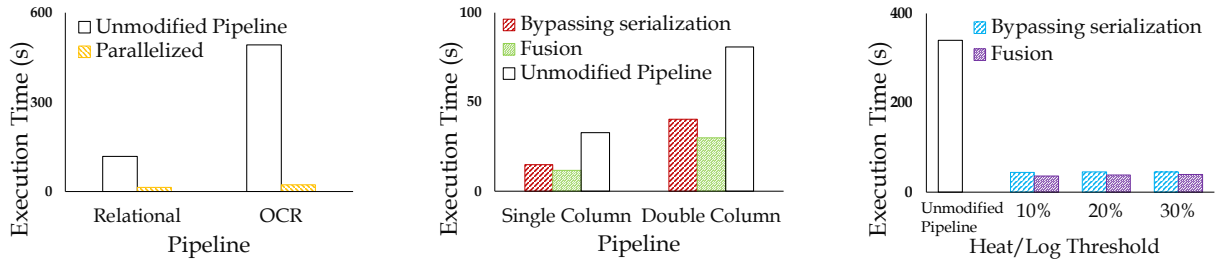
Rewrites to avoid serialization. When two points are deemed inverse, Caesar modifies the applications to bypass the serialization and deserialization by replacing these calls with writing and reading the raw binary value directly. To avoid misinterpreting serialized data as binary ones or vice versa, Caesar synchronizes the two applications before switching to binary intermediary data – currently using the number of produced and consumed data. Furthermore, fusion plays an important role in state-of-the-art analytical engines. Caesar extends points matching into partial application fusion: when a single serialization unit is fully matched, such as in the case of the Int-Sum pipeline, Caesar fuses the two applications into one executable to further reduce the communication overhead.

7 Runtime Instrumentation

Source-to-source transformation. Caesar uses source-to-source translation, namely the *removal* of calls to (de)serialization functions. This allows Caesar to modify the utility behavior near the application boundaries, while maintaining the pipeline semantics. So from a single-application point of view, the optimized utility produces or consumes a different data format, and thus it is not equivalent to the initial program. It also allows for language implementation details, for example if a given language is statically or JIT compiled, to remain transparent to Caesar. Caesar implements the source-to-source transformations in ANTLR4 [29], a parser-generator library that ingests grammar specifications and generates parse-tree visitors. The generated ANTLR visitors are specialized to perform the appropriate source code modifications, such as omitting the code generation for specific nodes. ANTLR grammars are maintained for many programming languages, so the cost of adding a new languages to Caesar is limited to visitor specialization.

Avoiding versus replacing serialization. Caesar's optimizations to avoid serialization are opportunistic: Caesar instruments the utilities to prevent calling expensive data transformations when it detects that the internal format matches the two applications. However, this can miss two important cases. First, the native data types across the two applications may differ (e.g., C++'s int and Python 3's variable size integers). To handle such cases, Caesar would have to marshal integers from the producer to the consumer in a data-type specific manner – essentially requiring Caesar to understand the binary representation and equivalences across multiple languages. Second, the two applications use different data types for the same variable (e.g., the consumer uses a 64bit integer while the producer uses a 32bit integer). To handle such cases, Caesar would have to lift the data serialization code and insert casts across data types – essentially requiring Caesar to understand logical typecasting. Essentially, in both cases, the additional benefits stem from replacing serialization points – instead of avoiding them and shortcircuiting a pre-serialization point with a post-deserialization point. However, both cases require lifting [5, 6] and weighing the replacement gain, which opens a new research direction. In Caesar, we rely on simple binary equivalences and exact matches to prune the matching and discovery effort – showing GOLAP's potential even when using simplistic serialization avoidance.

Data shuffling. Caesar currently supports one producer one consumer. In the cases where there is more than a single producer to consumer, Caesar can be expanded to identify the data flow from the pipeline description and apply the appropriate modifications. In order to have a single-producer, multiple-consumer pipeline, pipelines will include a splitting utility like `tee`. Caesar can scan the pipeline for known split operations, inject logging for each



(a) Orchestration **(b) Avoiding serialization** **(c) Combined optimizations**

Figure 3: Impact of different optimizations performed by Caesar

operator, and do an n-way comparison of logs since the split can be statically detected before instrumentation begins. In the case where we have n instances of the producer and n instances of the consumer (potentially as the result of parallelization), Caesar currently ensures that the logs are process-private.

Rewrite granularity and state. Caesar performs rewrites in multiple stages. Two factors affect rewrite order: whether the number of application instances changes and how the state is maintained across rewrites. Specifically, 1) splitting a running application into two new independent tasks is a non-trivial operation, and 2) restarting a running application without losing the internal state requires, in the general case, support by the application. Caesar takes advantage of the process-level execution model in data pipelines to avoid such demanding requirements. Specifically, parallelization and utility fusion change the number of application instances. Thus, Caesar decides parallelization and fusion optimizations a priori to instantiating the pipeline – avoiding the requirement for task splitting and consolidation of two running tasks into one process.

In contrast, avoiding serialization does not change the number of launched applications, providing an additional opportunity for maintaining state across rewrites. Instead of rewriting applications offline and restoring the state, we can do the rewrites online and keep the state without additional effort. A naive approach to rewrite an application to avoid a serialization point would recompile and restart the application, losing the internal state. Caesar only opts for such an offline approach, which restarts the application after a rewrite, for rewrites between two consecutive pipeline invocations. However, Caesar also supports *online* serialization-avoidance rewrites during the execution of a pipeline. To enable online rewrites, Caesar exploits the fact that modifying the (de)serialization code does not affect internal data structures. Thus, Caesar uses just-in-time code modification to replace the serialization without affecting the surrounding code.

8 Evaluation

We evaluate Caesar using two microbenchmarks per proposed optimization and a combined-optimizations benchmark as an end-to-end evaluation on a multi-utility pipeline.

Experimental setup. The experiments run on a dual-socket Intel Xeon E5-2650L v3 CPU machine with 12 physical cores per socket and a total of 256GB memory. We use dockerized Ubuntu 20.04.2, GNU Coreutils 8.28, Bash 5.0.17, and JVM Corretto-11.0.10. Caesar is written in C++ and invokes ANTLR 4.3 to modify inspected utilities. Utilities are given in the form of compiled binaries. In order to apply serialization optimizations, the binaries must be accompanied by their (C/C++) source code and the compilation command. Binaries without accompanying source code can still be run and parallelized. Caesar receives the input pipeline as a parsed representation of a simplistic scripting language. All files are considered immutable and all inter-utility communication happens through stdin/stdout. For the evaluation, binary modifications are performed offline, and we report the corresponding offline time.

Microbenchmark: orchestrating utilities. To evaluate the parallelization opportunities, Figure 3a uses: i) "Relational", a relational-like pipeline that scans and ranks a set of text files based on a keyword frequency, ii) "OCR" uses Tesseract [16], an OCR library, to perform the same keyword search and rank over images. Parallelizing the Relational text search achieves 14.5x speed-up over the sequential pipeline description, while the computationally heavier OCR pipeline has speed-up close to the number of physical cores.

Microbenchmark: optimizing utility boundaries. Figure 3b evaluates the benefit of avoiding inter-utility serialization, using two pipelines: i) "Single column", a number generator that pushes one column to a summation, ii) "Double column", an <IP, size> generator pushing to an aggregator to find who sent the most data.

The first optimization "Bypassing serialization" applies only serialization avoidance, without fusion or parallelization, achieving approximately 2x speed-up over the baseline, unmodified pipeline, for both pipelines. The second optimization, "Fusion" additionally performs fusion for the selected utilities, pulling the code of the second utility into the first one, achieving up to 2.8x speed-up, as it avoids sending even binary data across the two utilities as well as invocation overheads.

Caesar employs logging to determine whether a pair of candidate serialization points is a match across utilities. For the Single and Double column pipelines, we measure a maximum 17% slowdown during the logging window compared with the baseline unmodified pipeline. For more complex pipelines, however, the slowdown further decreases as the pipeline complexity increases, giving only 1% overhead in the case of the "combined optimizations" experiment. Furthermore, inserting logs and bypassing serialization require modifying and recompiling utilities, taking 2-3 seconds per binary.

Combined optimizations. Figure 3c evaluates the potential of Caesar in a more complex pipeline: the NOAA pipeline, from Figure 1, modified for local-only files and using `ls` instead of the loop. For the corresponding bars in the figure, we trigger logging at 10/20/30% hotness and rewrites after inspecting 10/20/30% of the final data through the logs. Due to the multitude of heavy operations, like unzipping and regex processing, there are relatively small savings by avoiding the serialization. Avoiding the serialization, however, enables fusion which has a higher impact. Overall, when all optimizations are enabled, Caesar achieves ~17x speed-up over executing the pipeline using its original description, by combining all the optimizations to take advantage of the multi-core CPU, the task parallelism created by the multiple files, and the saved operations by the utility fusion.

9 Conclusion

Traditional DBMS architectures are designed to support a closed set of operations determined and implemented by the database vendor, with little support for efficient execution of arbitrary external functions other than black-boxed UD(A)Fs. In this work, we make the case for a new DBMS architecture paradigm that places the DBMS into an orchestrating role that gradually understands and adapts to its surrounding ecosystem through runtime inspections and code modification. As a result, GOLAP allows optimized execution of data pipelines that use utilities initially unknown to the DBMS, with performance similar to integrated functionality with minimal user effort. Finally, we show through microbenchmarking a proof of concept for the applicability of the proposed design for simplified programs and demonstrate a potential 22x speed-up by modifying the utilities just-in-time.

Acknowledgments

This work was partially funded by the EU H2020 project SmartDataLake (825041) and the SNSF project "Efficient Real-time Analytics on General-Purpose GPUs" subside no. 200021_178894/1.

References

[1] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999.

[2] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005.

[3] M. J. Cafarella, D. J. DeWitt, V. Gadepally, J. Kepner, C. Kozyrakis, T. Kraska, M. Stonebraker, and M. Zaharia. A polystore based database operating system (DBOS). In V. Gadepally, T. G. Mattson, M. Stonebraker, T. Kraska, F. Wang, G. Luo, J. Kong, and A. Dubovitskaya, editors, *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB Workshops, Poly 2020 and DMAH 2020, Virtual Event, August 31 and September 4, 2020, Revised Selected Papers*, volume 12633 of *Lecture Notes in Computer Science*, pages 3–24. Springer, 2020.

[4] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[5] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. Statusquo: Making familiar abstractions perform using program analysis. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[6] A. Cheung, N. Crooks, J. M. Hellerstein, and M. Milano. New directions in cloud programming. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[7] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.*, 12(5):544–556, 2019.

[8] K. Dursun, C. Binnig, U. Çetintemel, G. Swart, and W. Gong. A morsel-driven query execution engine for heterogeneous multi-cores. *Proc. VLDB Endow.*, 12(12):2218–2229, 2019.

[9] C. Duta, D. Hirn, and T. Grust. Compiling PL/SQL away. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[10] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In H. Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, pages 102–111. ACM Press, 1990.

[11] M. Greenberg, K. Kallas, and N. Vasilakis. *Unix Shell Programming: The next 50 Years*, page 104–111. ACM, New York, NY, USA, 2021.

[12] A. Heinricher, R. Williams, A. Klingbeil, and A. Jordan. *Weldr: Fusing Binaries for Simplified Analysis*, page 25–30. ACM, New York, NY, USA, 2021.

[13] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[14] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, 2016.

[15] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-in-time data virtualization: Lightweight data management with vida. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[16] A. Kay. Tesseract: An open-source optical character recognition engine. *Linux J.*, 2007(159):2, July 2007.

[17] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11:2209–2222, 2018.

[18] P. Kraft, A. Waterland, D. Y. Fu, A. Gollamudi, S. Szulanski, and M. I. Seltzer. Automatic parallelization of sequential programs. *CoRR*, abs/1809.07684, 2018.

[19] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. A. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. MLIR: scaling compiler infrastructure for domain specific computation. In J. W. Lee, M. L. Soffa, and A. Zaks, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 2–14. IEEE, 2021.

[20] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 743–754. ACM, 2014.

[21] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, Sept. 2017.

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

[23] S. Palkar, J. J. Thomas, D. Narayanan, A. Shanbhag, R. Palamuttam, H. Pirk, M. Schwarzkopf, S. P. Amarasinghe, S. Madden, and M. Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[24] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, page 16–27, New York, NY, USA, 1990. ACM.

[25] D. Raghavan, S. Fouladi, P. A. Levis, and M. Zaharia. Posh: A data-aware shell. *login Usenix Mag.*, 45(4), 2020.

[26] P. Sioulas, V. Sanca, I. Mytilinis, and A. Ailamaki. Accelerating complex analytics using speculation. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021.

[27] A. TAKANO. Y.futamura : Partial evaluation of computation process : An approach to a compiler-compiler. *IPSJ Magazine*, 43(12):1395, dec 2002.

[28] O. Tange. GNU parallel: The command-line power tool. *login Usenix Mag.*, 36(1), 2011.

[29] Terence Parr et al. Antlr.

[30] N. Vasilakis, K. Kallas, K. Mamouras, A. Benetopoulos, and L. Cvetkovic. Pash: light-touch data-parallel shell processing. In A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 49–66. ACM, 2021.

[31] A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. I. Seltzer. ASC: automatically scalable computation. In R. Balasubramonian, A. Davis, and S. V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 575–590. ACM, 2014.

[32] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.