EUROGRAPHICS 2015 / H. Carr, K.-L. Ma, and G. Santucci
(Guest Editors)

Volume 34 (2015), Number 3

Perfopticon: Visual Query Analysis for Distributed Databases

Dominik Moritz, Daniel Halperin, Bill Howe, and Jeffrey Heer

Computer Science & Engineering, University of Washington

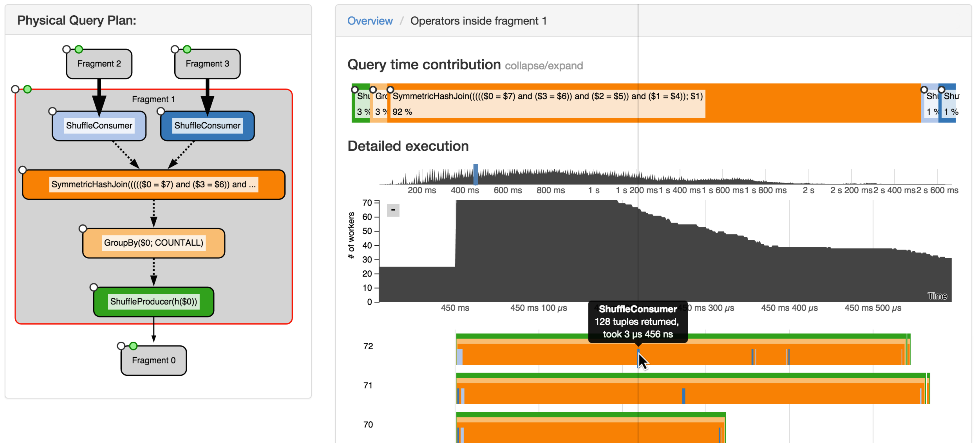


Figure 1: Perfopticon visualizing a recorded query execution that estimates species abundance in an oceanography dataset. The left panel displays the optimized query execution plan. The right panel includes (top) a divided bar chart showing the runtime contribution of individual query operators, and (bottom) a focus+context view of detailed per-worker execution traces.

Abstract

Distributed database performance is often unpredictable due to issues such as system complexity, network congestion, or imbalanced data distribution. These issues are difficult for users to assess in part due to the opaque mapping between declaratively specified queries and actual physical execution plans. Database developers currently must expend significant time and effort scanning log files to isolate and debug the root causes of performance issues. In response, we present Perfopticon, an interactive query profiling tool that enables rapid insight into common problems such as performance bottlenecks and data skew. Perfopticon combines interactive visualizations of (1) query plans, (2) overall query execution, (3) data flow among servers, and (4) execution traces. These views coordinate multiple levels of abstraction to enable detection, isolation, and understanding of performance issues. We evaluate our design choices through engagements with system developers, scientists, and students. We demonstrate that Perfopticon enables performance debugging for real-world tasks.

1. Introduction

As data acquisition continues to outpace data analysis in many fields, scientists and analysts are increasingly adopting distributed database systems and performing computation using ad hoc analytical queries over large datasets. Computation in a database is expressed in a declarative query language (often a variant of SQL) and then translated into a physical query plan for execution. This physical plan describes a distributed data flow that performs the local computation and moves records from machine to machine in the cluster as needed.

Although a declarative language helps to raise the level of abstraction for performing data analysis, there is a mental mismatch between high-level declarative queries and low-level physical plans that are chosen by the optimizer and executed by the database system. The translation from query to execution plan is opaque and makes it difficult for both developers and users to understand and diagnose unexpected system behavior, including poor performance, incorrect results, or failures [RH05, AB04].

Non-distributed databases are already highly complex systems, but distributed databases introduce significant new complexities: users must reason about distributed correctness, complex data flow patterns between machines, and new failure modes. These *query analysis* tasks cause database developers and users alike to spend significant time and effort isolating and debugging unexpected behavior and unexpected performance. Specifically, we find that users need to complete the following tasks (discussed in §4):

T1: Understand the specific computations being performed, after translation into a **query plan**.
T2: Obtain an **overview** of the work distribution to identify bottlenecks in the query plan.
T3: Understand distributed **communication** patterns to diagnose bottlenecks resulting from network effects.
T4: **Trace local execution** on each machine to diagnose bottlenecks resulting from non-distributed computation.

In the status quo, database developers and users process log files with ad hoc methods to compute system performance metrics (*profiling*) as part of query performance debugging. These log files are distributed across the cluster, are represented in non-standard formats, and include an enormous amount of irrelevant information. We argue that interactive visualizations designed specifically to support query analysis tasks should be considered a first-class design requirement in modern distributed systems.

In this paper we contribute Perfopticon, an interactive visualization tool for understanding both data flow and run-time performance in a distributed database. Perfopticon visualizes query execution logs in coordinated visualizations at different levels of abstraction to enable efficient profiling and debugging. Using our tool, we find that both database developers and users can quickly discover run-time bottlenecks and problematic data imbalances, saving them significant guesswork and programming effort. Perfopticon includes the following components to support the query analysis tasks above:

- *Query plan view*: An interactive graph showing the query plan, which provides a starting point for further explorations (Figure 1, left).
- *Work distribution overview*: Small-multiple area charts showing cluster utilization over time by each query operator. In this view, a long tail identifies a worker that lags behind the rest of the cluster.
- *Communication view*: A matrix diagram that shows how much data was sent between each pair of workers. This view reveals patterns in the communication such as skew.
- *Local execution view*: A novel timeline chart visualization that compactly shows nested execution traces across workers (Figure 1, right). This view serves to identify and explain low-level problems in the local execution.

Perfopticon follows the common “overview first, zoom and filter, details on demand” navigation pattern [Shn96]. The interface begins with high-level overviews of both the query plan and cluster-wide performance details. Based on observed patterns, users can then drill-down by filtering performance views or navigating to more localized visualizations.

We developed Perfopticon in collaboration with the developers of Myria [HdAC*13], a distributed, shared-nothing [Sto86] big data management system. Although built for Myria, Perfopticon’s design easily generalizes to other operator-based data flow systems such as Hadoop, Hive [TSJ*09], Drill [HN13], and Shark [ELX*12], as well as commercial systems like Vertica [HP].

Beyond the system itself, this paper contributes domain specific and task-relevant abstractions for visual query analysis, and “lessons learned” during the design of a scalable visualization system.

To evaluate Perfopticon, we performed a long-term deployment (6 months) with database researchers (including two paper authors) and conducted informal studies with Myria users. We found Perfopticon valuable for database development, query writing & debugging, and teaching. Participants successfully used Perfopticon to resolve performance issues, optimize queries, understand new distributed join algorithms, and teach basic operating principles of distributed databases.

2. Related Work

To allow users to tune their queries, database systems such as SQL Server, MySQL, and PostgreSQL let users obtain the query plan with the `EXPLAIN` keyword in SQL. The query plan is annotated with expected costs so that expert users can tune indexes and assess how different expressions of logically equivalent queries may result in different execution strategies. `EXPLAIN ANALYZE` annotates the query plan with the actual runtime and number of rows by profiling the query. The Microsoft SQL Server query plan visualizer [Mic] graphically presents the result of `EXPLAIN`. The Vertica Query Analyzer (VQA) [SWBW14] extends this idea to distributed databases. However, unlike Perfopticon these tools do not expose fine-grained execution traces, limiting users’ ability to understand the causes of observed behavior. Perfopticon’s query plan view builds on prior work on query plan visualization (T1).

Profiling tools for imperative programs, such as GNU gprof [GKM82], are widely used by software developers to analyze CPU and memory usage on a single machine. Interactive tools allow developers to explore vast amounts of profiling data. For example, the Chrome performance profiler [Goo] lets users interactively explore JavaScript execution, rendering, and network usage. Perfopticon’s local execution view builds on insights from these existing tools, but adapts them for use in the context of distributed databases.

Performance debugging is more challenging for distributed systems, and is largely unaddressed by conventional software debugging tools. Google uses Dapper [SBB*10], which maps individual call traces through their distributed systems. Users can explore collected traces in an interactive visualization. The scientific computing community developed tools to trace MPI (Message Passing Interface) events and calls in high-performance computing. Current visualization tools for traces such as Vampir [NAW*96], ParaGraph [HF03], and Paraver [FZB*08] use timeline charts with bars that are grouped by process and span the lifetime of events. Recently, Isaacs *et al.* presented Ravel [IBJ*14], which uses logical time to unravel overlapping lines that depict communication between events in MPI. Parallel operator-based data flow systems, such as Myria, regularly shuffle data between each pair of machines. Hence, causalities and direct communication are less relevant, impeding the direct application of Vampir and Ravel. Moreover, distributed databases process large data resulting in many traces from only a few operators. Perfopticon uses a timeline chart to show local execution (T4) in each fragment but focuses on operators instead of traces and hides direct communication. We applied the matrix view from Vampir (showing message exchanges) in Perfopticon to show how many tuples have been sent between fragments.

In pure trace debugging tools, developers can lose the context of the original computational abstractions such as the query plan, and are forced to view the computation solely as a set of parallel event streams. Offering higher-level abstractions for domain-specific concepts can dramatically simplify developer tools and provide necessary context. Twitter's Ambrose [Twi13] is a platform to visualize and monitor MapReduce workflows using views of associated jobs, job dependencies, and job progress. However, this approach is not suitable for our needs, as "jobs" constitute too coarse of an abstraction: they do not reveal individual operators as they appear in database query execution plans.

In Perfopticon, we build on the insights from systems such as the SQL Server query plan visualizer, the Chrome profiler, Dapper, Vampir, and Ambrose. We combine and extend these ideas in a novel UI for analyzing distributed operator-based data flow computations, informed by visualization design principles [Bre99, CM84, HR07, JME10].

3. Background: Distributed Query Execution

The design of Perfopticon was originally motivated by Myria, a state-of-the-art distributed database system typical of a large class of systems to which Perfopticon's design can apply. As shown in Figure 2, Myria has a single *master*, which is a server responsible for optimizing the query and coordinating query execution. The query is executed by a set of workers that can communicate with each other.

A user (Figure 2a) writes a query in a declarative language, which is sent to the master. On the master, the optimizer (Figure 2b) first translates the query into a *logical query plan*, which is a tree of relational algebra operators such as selections, projections, and joins. Next, the logical plan is optimized according to known rules, such as moving selection operators earlier in the dataflow to send as little data over the network as possible. The optimized logical plan is then translated to a physical query execution plan, or *query plan* (Figure 2c), consisting of physical operators. A logical join operator could be instantiated as a `MergeJoin` or a `HashJoin`. These optimization steps require knowledge about the data in the system, their schemas, and statistical properties, which are stored in the catalog (Figure 2d).

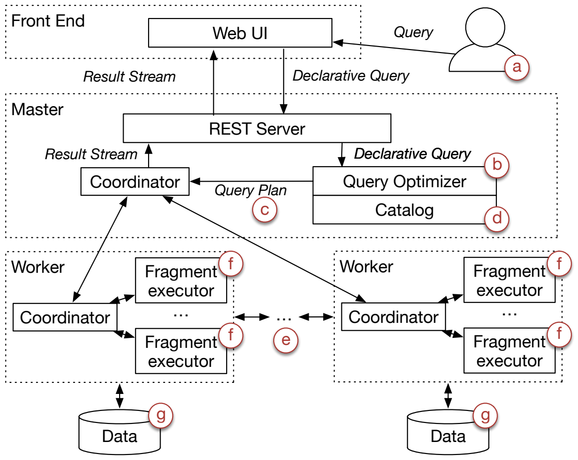


Figure 2: Overview of the Myria architecture. A query is sent through the front-end to the master; the optimizer translates it to a physical query execution plan. The query execution plan consists of parts that can be executed locally on each machine (fragments) with communication steps in between.

Myria is a parallel data flow evaluation system that takes a graph of operators and executes it on a shared-nothing [Sto86] cluster (example query plan in Figure 3). Execution spans multiple machines and operators are processed in *parallel*. Flow control must ensure that operators start computing when data is available and wait when a downstream operator is backlogged. In a *shared-nothing* system, all data is partitioned across workers: each server is responsible for a certain part of the keyspace and each tuple (row) in the database belongs on exactly one worker. Thus, if data has to be aggregated or joined on a key by which the data is not partitioned, data must be repartitioned (*shuffled*) according to a new key, using a hash function from key to worker. These shuffle steps require inter-worker communication (Figure 2e) over the network, but all other computations can be local and independent.

In Myria, a *fragment* is a set of local operators that is independent of global flow control. Fragments split the query plan into distinct subtrees of operators that can be executed on many workers in parallel (Figure 2f). The final operator in a fragment (the root of a fragment's tree) synchronously requests data from its children. The initial operators (the leaves of the tree) receive data either asynchronously from other workers or synchronously from a local data store (Figure 2g). When called by its parent, each operator either returns a complete batch of tuples, an incomplete batch, or a signal that all data has been processed. An operator may repeatedly call its children for more data. The fragment as a whole will cease computation ("go to sleep") when each of its operators has processed all currently available data. It will "wake up" when new data is received by the worker it is running on.

To investigate performance issues, it is sometimes necessary to examine both data partitioning and data flow and to understand local execution inside of fragments. For example, to process a distributed join, tuples that share a key are sent to the same worker. If there is a popular key, certain workers will receive disproportionately large amounts of data to process. This *skew* could also be caused by a hash function assigning too many keys to the same worker. Other causes of *stragglers* (workers that finish long after other workers) include failures at the hardware or OS level and non-uniform computational complexity across workers. Stragglers delay dependent computation and hurt performance, as total query runtime is determined by the slowest worker.

4. The Design of Perfopticon

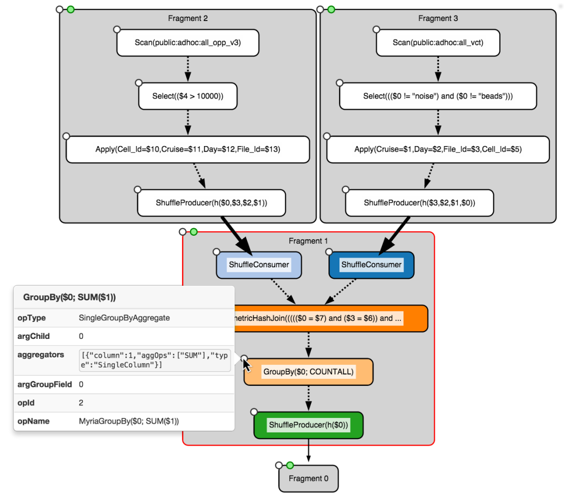
In this section, we describe the constituent components of Perfopticon, and justify our design decisions based on system restrictions, best practices, and results from interviews.

The four tasks users wished to accomplish when analyzing queries from §1 are informed by formative interviews with developers and users, experience with prototypes, and observations of common debugging practices. Query debugging (T1) is long recognized as difficult due to the dissimilarity between query and query plan [Har10, RH05]. We found that performance issues are caused by bugs, missing optimizations, an incorrect query, problems with the servers, skewed communication, or imbalanced data distribution. Hence, debugging often starts without a clear hypothesis. An overview (T2) over the whole execution provides a way to narrow down the source of an issue, which also helps delivering the tool to non-experts. It is well known that communication (T3) is a major bottleneck in data-intensive computing [XKZC08, DG92]. Fine-grained debugging and profiling requires inspection of the low-level execution (T4). Initially, we expected summaries to be sufficient but early user experiences revealed a strong need for fine-grained debugging.

Perfopticon was developed through an iterative user-centered design process. Based on the identified tasks (T1 to T4), we sketched interfaces and developed visualizations based on data from log files. We then developed a prototype that collected log files from all workers on the master and transformed the log data into a suitable format for the front-end. Based on the feedback from the first prototype, we then focused on scalability and simplified the visualizations by removing redundant data. In a third pass, we refined features and improved system performance.

In the latest version, Perfopticon's interface consists of the *query plan view* on the left and the *details panel* on the right (Figure 1). The query plan view always shows a graph representation of the query plan and the details panel shows either a work distribution overview, communication view, or local execution view. Above the details panel is a breadcrumb navigation, which allows the user to go back to the overview.

We describe Perfopticon via a usage scenario in which Emma, an oceanographer, writes a complex query to estimate the abundance of *synechococcus*, a common marine cyanobacterium, in data collected by ocean-deployed sensors. Modeled on a real Myria user, Emma is both an expert in her field and a Myria "power user": she understands the execution model, has expectations about performance, and performs follow-up analysis when her expectations are not met.


Fragment 2
Scan(public:adhoc:all_opp_v3)
Select(($4 > 10000))
Apply(Cell_Id=$10,Cruise=$11,Day=$12,File_Id=$13)
ShuffleProducer(h($0,$3,$2,$1))
Fragment 3
Scan(public:adhoc:all_vct)
Select((($0 != "noise") and ($0 != "beads")))
Apply(Cruise=$1,Day=$2,File_Id=$3,Cell_Id=$5)
ShuffleProducer(h($3,$2,$1,$0))
Fragment 1
ShuffleConsumer
ShuffleConsumer
netricHashJoin(((($0 = $7) and ($3 = $6)) and ...
GroupBy($0; COUNTALL)
ShuffleProducer(h($0))
Fragment 0

GroupBy($0; SUM($1))

opType	SingleGroupByAggregate
argChild	0
aggregators	[{"column":1,"aggOps":["SUM"],"type":"SingleColumn"}]
argGroupField	0
opId	2
opName	MyriaGroupBy($0; SUM($1))

Figure 3: Query plan view for a query with a two-way join and an aggregate. The user collapsed Fragment 0 and selected Fragment 1 to view details in the right side view. The edges going into Fragment 1 are wider than the outgoing edge, indicating that the number of tuples is decreasing.

In this example, Emma must join multiple datasets: one with biological information and one with environmental information. When she submits the query, she observes that her query is running slower than expected. She hypothesizes that there could be a mistake in her query, a transient problem in the cluster, or even a bug in the database itself. Emma uses Perfopticon for query analysis through the four tasks of understanding the query plan (T1), overall work distribution (T2), communication (T3), and local execution (T4).

4.1. Query execution plan view

Emma inspects the query plan, finding that a join was translated to a cross product with output quadratic in the size of the input. Realizing that the problem arose from a mistake in her query, she fixes the query and tries again (Figure 3).

The left panel always shows a graph of the query plan, which reveals the result of the decisions made by the optimizer (T1). It helps users orient themselves and provides a starting point for further exploration.

Figure 3 shows Emma's revised query plan. Fragments are shown as groups around operators. Operators are shown as nodes, each labeled with both the name of the operator and the most important parameters to distinguish operators of the same type (e.g., the `Scans` in Fragment 2 and 3 in Figure 3). In the top-left corner of each node is a white circle, which provides details on demand about the operator or fragment. The

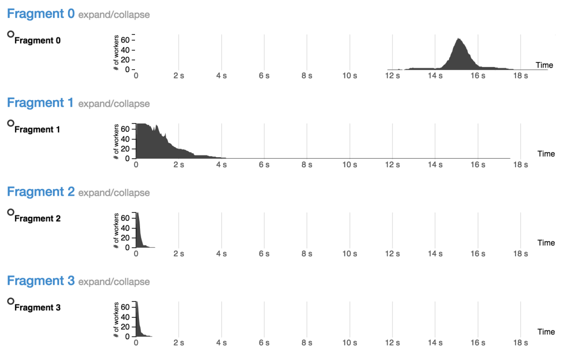


Figure 4: Overview showing small multiples of utilization over time, grouped by fragment. Utilization is measured as the fraction of workers executing each fragment. The long tail in the utilization chart for Fragment 1 shows that a few workers take significantly longer, indicating skew.

details about operators vary from type to type and show the complete set of parameters for the operator, *i.e.* the columns to aggregate and the aggregate function (Figure 3, tooltip).

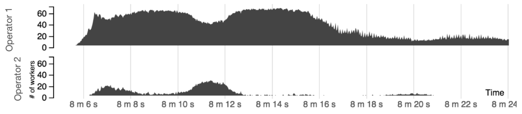
An edge between two operator nodes indicates a parent-child relationship; the parent operator receives tuples from its children. As explained in §3, tuple-passing can result either from a synchronous call to an operator or asynchronous input from the network. Edges involving network communication provide an initial hint for expensive shuffling steps since they have different widths depending on the number of tuples sent (exact numbers in tooltip). The smallest width maps to no tuples and the largest width to the maximum number of tuples (we highlight zero tuples).

Analytic queries are often complex and the resulting plans consist of many operators and fragments. A representative large query in Myria has 40 fragments and up to 16 operators (with a depth of less than 5) in a fragment. In Perfopticon, fragments are automatically collapsed if the query plan has too many fragments. A user can expand and collapse fragments using the green circle in the top left corner of fragment nodes. Transitions are animated to facilitate perception of changes between layouts [HR07]. The query plan view can be zoomed by scrolling in the left view area.

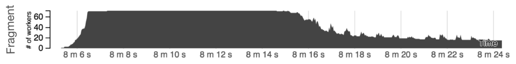
4.2. Overview of Work Distribution in all Fragments

Although Emma fixed her query, she notices that performance is still not what she expected. She uses the overview of work distribution (Figure 4) to investigate if there is an issue with a worker in the cluster. She looks for stragglers (unusually slow workers), which could suggest problems with the cluster itself. Emma finds that there are in fact stragglers caused by an overloaded machine in the cluster.

The default view shown in the details panel is an overview of usage over time across all fragments. It provides a summary of the global execution and aids comparison between the execution in different fragments (T2).



(a) Utilization broken down by operator.



(b) Summarized for whole fragment.

Figure 5: The fraction of workers executing operators in a fragment (top), and the same data summarized as the fraction of workers executing any operator in the fragment (bottom).

The overview is a small-multiples view of time series area charts, each showing the fraction of workers executing an operator over the lifetime of the query. Inside a fragment, only one operator is executed at a time while the fragment is active. As shown in Figure 5, we can aggregate the area charts for each operator in a fragment into a single area chart showing the fraction of active fragments across all workers. When a fragment is expanded in the query plan view, the utilization is broken down by operator. Otherwise we show the aggregated data.

We chose small multiples rather than stacked area charts or overlapping line charts because of their superior performance on comparison tasks [JME10, BW08]. The labels to the left of each area chart show the operator name, slightly indented based on depth in the operator tree.

In Figure 4, long tails in the area charts reveal skewed execution, where a few workers finish long after other workers. Spikes are an indication of cluster-wide synchronization caused by similar execution times or timeouts. We can see when certain fragments finish well before others, and which operators are run at different stages of the query execution. In Emma's query the Fragments 2 and 3, which read data from disk and shuffle it, finish well before the other fragments.

The user can zoom in on a time range in the small-multiples views by brushing within any of the charts. The zoom is animated to preserve context [HR07]. Immediately upon zoom, the available data is interpolated to provide a coarser level-of-detail. Later, the view is updated with higher resolution data. This technique does not prevent update lag (which can negatively affect visual exploration [MW93, LH14]), but does let users continue their visual exploration uninterrupted and without losing context.

4.3. Network communication view

After Emma frees resources on the overloaded machine, the query improves but is still slower than expected. She uses Perfopticon to reevaluate query execution. She now notices a single slow running worker in the overview visualization and disproportionately large amounts of data coming from Fragment 1. This sparks her interest and she clicks on the edge between fragments to open the communication view

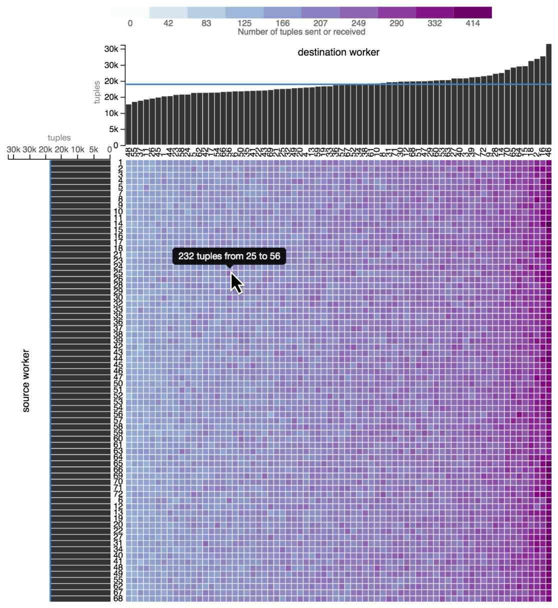


Figure 6: Communication matrix for a join query on a Twitter follower graph, running over a 72-worker cluster. Source workers are on the y-axis and target workers on the x-axis. The matrix shows that all source workers send the same number of tuples, but that the amount of data sent to target workers is skewed due to large differences in follower count among Twitter users.

in the details panel. She notices disproportionately large amounts of data sent from one worker.

In Perfopticon a user may click on any edge between fragments in the query plan to investigate communication related issues (T3) in the communication view. This view reveals communication patterns such as skew (§3). In a previous iteration, this view showed communication over time for an edge, but it did not help our users to gain new insights.

The central visualization of this view is a communication matrix visualizing the number of tuples sent between every pair of workers (Figure 6). The source worker is on the y-axis and the target worker on the x-axis. We chose to show the number of tuples instead of bytes, as tuples (or rows) are the atomic unit of most data flow systems. We used Cynthia Brewer's `BuPu` palette [Bre99] to encode the number of tuples sent. Tooltips show exact values.

Marginal histograms along the x- and y-axes show the count of tuples sent/received, overlaid with an average line to assist comparison. Users also asked us to support reordering the columns to place outliers closer together. Perfopticon supports ordering by worker id and number of tuples shuffled, covering the use cases we have encountered thus far.

We use a matrix diagram because it compactly visualizes bidirectional communication between all workers. We considered alternative designs such as a circular chord diagram [KSB*09] (as used by Twitter Ambrose [Twi13]), but found that chord diagrams do not scale well when visualizing large, dense matrices: in a chord chart each communication requires drawing a link and in a matrix diagram only a simple rectangle. The matrix diagram in Perfopticon scales well up to 100 workers, at which point labels begin to overlap.

4.4. Local Execution Trace View

Drilling down into the local execution view, Emma investigates the straggling worker and finds that the join operator produced more tuples than expected. In this case, a sensor took more measurements at a particular time and location. Based on this insight she can enable special rules in the optimizer that better distribute the work of the join.

The execution trace view presents detailed performance traces within a single fragment, enabling users to further investigate patterns (e.g., long tails) found in the overview (§4.2). This view is opened when a fragment is selected in the query plan view or the overview. It consists of a summary chart showing how much each operator contributed to the runtime, a focus+context area chart showing overall utilization, and a timeline showing nested execution traces (T4). The operators of the selected fragment are colored in the plan view using Tableau's categorical color palette, making the query plan a legend for other visualizations.

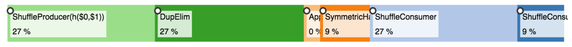


Figure 7: A divided bar chart illustrating each operator's proportional contribution to runtime. Circles at the top left of each bar provide details on demand about an operator.

At the top of this view a divided bar chart illustrates how much each operator proportionally contributes to the query execution time (Figure 7). We chose this chart over a pie chart because we have more horizontal space available in the interface and because the perception of angles generally has higher errors [CM84]. The bars are sorted by their depth in the operator tree. We experimented with a treemap showing the complete hierarchy, but our users reported it to be more confusing than helpful.

Below the contribution bar chart is a timeline showing detailed execution traces (Figure 1, right). These traces depict when a worker executes the selected fragment, along with details about which operators are called. The view consists of an area chart (with zoomable overview+detail) showing fragment utilization. Providing focus+context [CMS99] allows us to display detailed traces alongside the context of the overall execution. As before (§4.2), zooming is animated and Perfopticon's front-end requests data with a resolution that matches the number of horizontal pixels [LJH13].

Selecting a time range loads fragment execution traces into the canvas below the focus+context charts. Here, the user can compare execution across all implicated workers during the

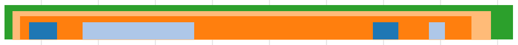


Figure 8: Timeline of nested calls involving five operators (from outer to inner): *Shuffle Producer* (●), *Group By* (●), *Symmetric Hash Join* (●), 2 × *Shuffle consumer* (● and ●).

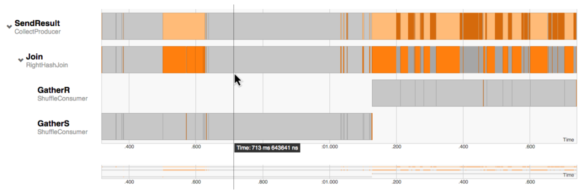


Figure 9: An early design for execution traces. Each operator occupies a separate lane. Colors indicate states: ‘active’ (●), ‘sleeping’ (●) and ‘waiting for child to return’ (●).

selected time range. The visualization shows when a worker (each in a separate lane) becomes active and what operators are executed while it is active.

Figure 8 provides an example of one such trace. First, the fragment executes the *Shuffle Producer* (●), which in turn synchronously calls its child *Group By* (●), which calls the *Symmetric Hash Join* (●). The Hash Join has two *Shuffle Consumers* (● and ●), which it calls one after the other. During the first call, a child returns tuples but the join can not produce a full batch. As the join operator’s children don’t have any tuples ready, the execution goes to sleep. The user may mouse-over the boxes to open a tooltip showing the name of the operator, the execution time and the number of tuples returned from the synchronous call. To help a user distinguish two adjoining calls to the same operator, we slightly expand the operator box to the bottom and give it a brighter color while the cursor is over the operator.

Initial designs (Figure 9) explicitly showed the states of each operator in different lanes. Our final design removes redundant information and is more compact, allowing users to examine traces from many workers. This design also exhibits better performance, as fewer bars are drawn. Developers found the fragment execution view useful to debug low level performance issues. However, novices unfamiliar with the execution model outlined in §3 did not utilize this view.

5. The Architecture of Perfopticon

We implemented Perfopticon on top of Myria, with only minor modifications to the Myria system itself. The front-end is web based and integrated into the Myria web front-end. In this section, we describe our implementation, including needed modifications to the back-end, and discuss how Perfopticon can be used with other operator-based data flow systems.

5.1. The Back-End Collects Logs and Executes Queries

We designed Perfopticon with minimal requirements for the database system to make it easier to generalize beyond Myria. Perfopticon requires only (1) the query execution plan, (2) information about the number and destination of tuples sent from a worker, and (3) logs when an operator called its child or returned a call to its parent. For (2) and (3), we had to add only 3 log statements to the generic operator code in Myria.

In our first prototype, we used standard Java logging to files. These had to be collected on the master, parsed, and the extracted data transformed into a suitable format for the visualizations. This design did not scale beyond second-long queries. In our final implementation, Perfopticon collects data during query execution with small runtime impact. Event logs are written to two special tables (one for tuples sent (2) and one for operator calls (3)) in the local database (Figure 2g). This “reflective” data design, storing the usage data as first-class relations within the database, enabled large gains in scalability (queries are parallelized) while simplifying the overall design (we reused Myria’s query execution logic). Performance data can be analyzed both via our visualizations and directly via SQL queries. The underlying data for every visualization in Perfopticon are computed via a small set of queries that filter and aggregate these two tables. For zoomable visualizations (*e.g.* Figure 4 and Figure 5), the resolution of the query result is typically the number of horizontal pixels. By manually optimizing the queries, we reduced lag and achieved interactive query response times of ~100 ms. For our tests, we used a cluster of up to 72 workers and queries running for ~20 min.

5.2. Implementation of the Front End Visualizations

Perfopticon’s front-end is implemented using Python and Google AppEngine on the server side and D3 [BOH11] on the client side. When a query is executed in Myria, logs are collected during execution and a link to Perfopticon appears when the query has finished. The query execution plan visualization (§4.1) takes the JavaScript Object Notation (JSON) description of the query plan, extracts operators and fragments, and computes the layout using Graphviz [EGK*02] (compiled to JavaScript with Emscripten [Zak11]). All other visualizations fetch data from the Myria REST server using asynchronous JavaScript (AJAX) as JSON or comma separated text (CSV) files.

To make the front-end scalable in the query runtime, all time series views use binned data at the available pixel resolution. The traces view (§4.4), which also scales with the number of workers, hides blocks that are shorter than one pixel and only shows root operators for long time ranges. The browser caches datasets to improve response rates when the user navigates back to a view she has seen previously.

6. Evaluation: Perfopticon Usage Examples

We demonstrate the efficacy of Perfopticon in use cases covering performance debugging, bug hunting, teaching, and research. These use cases come primarily from a 6 month deployment of Perfopticon within the Myria development team as well as usage by collaborating scientists leveraging Myria for data analysis. Both developers and users have given us strong positive feedback. For example, one user reported that "Lacking proper query plan (DAG) visualization and profiling is one of the main drawbacks of Spark [a popular distributed data analysis platform] compared with Myria" and "The number of things I was able to do with Perfopticon is huge." Perfopticon has thus far generally met our expectations, but also had positive unforeseen use cases.

We show these use cases to demonstrate that the visualizations in Perfopticon facilitate performance debugging for users (§6.1) and developers (§6.2, §6.3), algorithm design and development (§6.4), and understanding of parallel query execution by non-experts (§6.5).

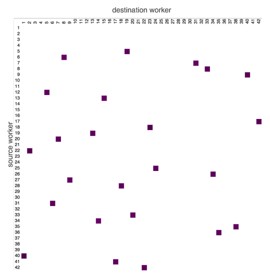
6.1. Identifying Performance Bottlenecks in a Cluster

In one instance, a query was running slower than usual and a user decided to profile the query. Perfopticon's overview showed a small number of workers taking significantly longer than the others, apparent as a long tail in the utilization chart. The user went to the fragment detail view and zoomed in on the long tail, filtering the execution traces to those active workers. Perfopticon enabled the user to determine that the problem was not a data skew issue (*e.g.*, the network view didn't show any abnormal communication), was not related to a single worker, and was not related to the whole cluster. Instead, the user discovered that all slow workers were on the same physical machine in the cluster. She resolved the issue by terminating a large process on the slow machine.

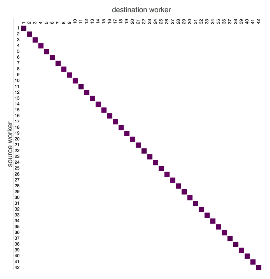
6.2. Implementing Deterministic Shuffling

A developer investigated an unexpectedly long-running query. The root cause was a redundant shuffle: *i.e.*, hashing on a field we already partitioned on. To debug this issue, he profiled the query and opened the communication view to discover unnecessary communication as seen in Figure 10a. Immediately he could see that all tuples from one worker were needlessly sent to another worker by a shuffle operator. Comparing multiple runs of the same query he discovered that the destination servers were different and as a result query runtime was inconsistent.

The underlying issue was a randomized (non-deterministic) shuffle. In response, the developer updated the mapping from hash value to worker to function in a deterministic manner. When the issue was fixed, the developer used Perfopticon to confirm that the issue is in fact resolved (Figure 10b). Now, the runtime is more stable and queries run up to 10× faster. In this case, Perfopticon sped up identification of the bug as well as confirmation of the fix.



(a) Tuples have to move between workers.



(b) Tuples stay on the same worker.

Figure 10: Communication for a redundant shuffle before (left) and after (right) implementing deterministic shuffling.

6.3. Debugging Incorrect Hashing in Aggregations

A scientist wrote a query to reimplement an existing R script. The script was roughly 100× slower, but well-tested. Though the query appeared correct, the results that Myria returned differed from the script. During debugging, a developer working with the scientist realized that the result actually changed every time; hinting at a concurrency or data order issue. Moreover, the issue only appeared for datasets multiple GB in size with queries taking longer than ~3 min.

The developer ran the query multiple times and compared the number of tuples exiting each operator across runs using the query plan and communication view. He found the problem to be in a fragment with an aggregate operator. The aggregate operator maintains a hash map of aggregators and merges incoming tuples with the same key. Somehow new tuples were not being added to an existing aggregator.

The developer found the bug in the custom hash table implementation, wherein updates upon hash collision were not handled correctly. This subtle bug only occurred at scale, as it required the right data interleaving and a hash collision. To get the hash collisions, about 2^{16} tuples are needed on the same machine. With a 72-worker cluster, a dataset of expected size $2^{16} \times 72 = 4.7M$ tuples is needed to replicate the bug. Perfopticon was helpful as it enabled the developer to investigate the issue at the required scale.

6.4. Analyzing a Skew-resilient Join Algorithm

A researcher evaluating different join strategies found that a new algorithm has better skew robustness, a previously unknown property. The researcher ran a query that finds the number of triangles (three users who transitively follow each other) in the Twitter follower graph. The query in SQL is:

```
SELECT a.follower, b.follower, c.follower
FROM twitter AS a, twitter AS b, twitter AS c
WHERE a.followee = b.follower AND
  b.followee = c.follower AND
  c.followee = a.follower
```

Traditional databases execute this join via cascading two-way joins, requiring two shuffle steps. The researcher had designed an algorithm that requires only one shuffle step [CBS15]. He profiled the traditional execution with Perfopticon and saw in the query plan view that the edge after the first join was thicker, indicating that the first join was producing many tuples that then need to be shuffled. This confirmed his expectation, as there are many join candidates in the Twitter follower graph. Upon drilling down into the network view, he found that the execution is highly skewed due to one extremely popular person in the dataset. He could also confirm that the imbalanced output of the first join is not due to imbalanced input by looking at the communication before the join.

The researcher then executed the same query with the new join algorithm that requires only one shuffle step. He discovered that fewer tuples had to be shuffled, and most importantly that the communication is evenly distributed across all workers. The researcher has since used plots from Perfopticon for their research paper on novel join algorithms [CBS15].

6.5. Using Perfopticon for Teaching

Perfopticon was used in a big data systems class to demonstrate how distributed query execution works. The teacher used the query plan view to show the translation of queries to an execution plan. He also used the communication view to illustrate skew. The fragment view with execution traces was used to explain how tuple batches are processed synchronously and how a fragment must be woken up when new tuples arrive at a worker. Outside the class, Perfopticon has been used to teach novices about query optimization. For example, an expert user showed the runtime impact of cross products that can often be avoided by rewriting a query.

7. Conclusions and Future Work

In this paper, we introduced Perfopticon, an interactive query profiling tool comprising visualizations of the query plan, work distribution across multiple machines, communication, and local execution traces. We showed that the visualizations and the back-end of Perfopticon are reusable, and enable users to efficiently analyze query execution at scale. Perfopticon enabled a number of advancements in real-world tasks: developers have fixed critical bugs, end users have refined queries to achieve better performance, researchers have investigated and verified novel algorithms, and teachers have used the system to explain how distributed database systems work.

Our experience with Perfopticon shows the value of visualizations for developers and users of big data systems. Users were able to debug issues at a scale impossible without visual abstraction. Visualization also lets users discover unexpected issues and hints users towards causes they had not considered. Hence, making system internals accessible through APIs and visualizations should be a key design goal for big data systems, not an afterthought. Our evaluations further reinforce the value of organizing and aggregating overwhelming amounts of (performance) data using domain-specific abstractions and task-relevant groupings. Nonetheless, we learned that aggregation alone is not sufficient and that users eventually need to drill down to individual operator executions. This came as a surprise as Myria is processing billions of tuples.

In retrospect, we found that a reflective design – using the database itself for storage and data transformation – simplifies application logic. Also, Perfopticon's back-end scales to large clusters and long-running queries since transformations are automatically parallelized.

Another observation is that building a visualization system for large data is a complex task. We implemented binning at the available pixel resolution (§4.4) to maintain client responsiveness. To support this, we had to implement complex aggregation queries and visualizations that can dynamically replace (and animate to) a different time range of the data. Existing visualization and programming frameworks do not support these tasks well.

Looking forward, Perfopticon can be extended to show more contextual information about the local execution, or even be integrated with a local debugger. In the next release, we will include overall CPU and memory usage in the cluster, and available/used network bandwidth in Perfopticon's UI. Moreover, more operator-specific metrics (*e.g.*, the size of hash tables in a join or aggregate operator) could be collected and shown in the interface on demand.

Unlike standard relational databases, Myria also supports iterative queries that enable users to run machine learning algorithms. Visualizing performance over asynchronous iterations (without global iteration bounds) adds another dimension of complexity. We are also interested in better ways to automatically label fragments by summarizing the operators they contain. Such summaries might also be used to label fragments when the query plan view is zoomed out [DGDGL07]. A current limitation of Perfopticon is the lack of support for comparison of multiple queries (other than using two instances of Perfopticon side by side). Such comparisons would simplify debugging issues like those described in §6.3.

Perfopticon is available as open-source software at `https://github.com/uwescience/myria-web`.

8. Acknowledgments

We thank the anonymous reviewers, the members of the UW Database Group & the Interactive Data Lab for their comments in improving this paper. We also thank Shumo Chu, Adriana Szekeres, Thierry Moreau, and Umar Javed for their help in developing the system. This work was supported in part by NSF Award 1247469, the Fulbright Program, the Intel Big Data ISTC, DARPA XDATA, the Gordon & Betty Moore Foundation, and the University of Washington eScience Institute.

References

[AB04] Avin C., Brito C.: Efficient and robust query processing in dynamic environments using random walk techniques. In *Proceedings of the 3rd international symposium on Information processing in sensor networks* (2004). 1

[BOH11] Bostock M., Ogievetsky V., Heer J.: D3: Data-driven documents. *Proc. IEEE InfoVis* (2011). 7

[Bre99] Brewer C. A.: Color use guidelines for data representation. In *Journal of the ASA* (1999). 3, 6

[BW08] Byron L., Wattenberg M.: Stacked graphs-geometry & aesthetics. *IEEE TVCG 14*, 6 (2008). 5

[CBS15] Chu S., Balazinska M., Suciu D.: From theory to practice: Efficient join query evaluation in a parallel database system. In *Proc. ACM SIGMOD* (2015). 9

[CM84] Cleveland W. S., McGill R.: Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the ASA 79*, 387 (1984). 3, 6

[CMS99] Card S. K., Mackinlay J. D., Shneiderman B.: *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999. 6

[DG92] DeWitt D., Gray J.: Parallel database systems: the future of high performance database systems. *Communications of the ACM 35*, 6 (1992). 4

[DGDGL07] Di Giacomo E., Didimo W., Grilli L., Liotta G.: Graph visualization techniques for web clustering engines. *IEEE TVCG 13*, 2 (2007). 9

[EGK*02] Ellson J., Gansner E., Koutsofios L., North S. C., Woodhull G.: Graphviz—open source graph drawing tools. In *Graph Drawing* (2002), Springer. 7

[ELX*12] Engle C., Lupher A., Xin R., Zaharia M., Franklin M. J., Shenker S., Stoica I.: Shark: fast data analysis using coarse-grained distributed memory. In *Proc. ACM SIGMOD* (2012). 2

[FZB*08] Funika W., Zientarski M., Badia R. M., Labarta J., Bubak M.: Performance visualization of grid applications based on ocm-g and paraver. In *Grid Computing* (2008), Springer. 2

[GKM82] Graham S. L., Kessler P. B., Mckusick M. K.: Gprof: A call graph execution profiler. In *ACM Sigplan Notices* (1982), vol. 17, ACM. 2

[Goo] Google: Performance profiling with the timeline. https://developer.chrome.com/devtools/docs/timeline. 2

[Har10] Haritsa J. R.: The picasso database query optimizer visualizer. *Proc. VLDB 3* (2010). 4

[HdAC*13] Halperin D., de Almeida V. T., Choo L. L., Chu S., Koutris P., Moritz D., Ortiz J., Ruamviboonsuk V., Wang J., Whitaker A., et al.: Demonstration of the Myria Big Data Management Service. 2

[HF03] Heath M. T., Finger J. E.: Paragraph: A performance visualization tool for MPI, 2003. 2

[HN13] Hausenblas M., Nadeau J.: Apache drill: interactive ad-hoc analysis at scale. *Big Data 1*, 2 (2013). 2

[HP] HP: Vertica. http://www.vertica.com/. 2

[HR07] Heer J., Robertson G. G.: Animated transitions in statistical data graphics. *IEEE TVCG 13*, 6 (2007). 3, 5

[IBJ*14] Isaacs K., Bremer P., Jusufi I., Gamblin T., Bhatele A., Schulz M., Hamann B.: Combing the communication hairball: Visualizing large-scale parallel execution traces using logical time. 3

[JME10] Javed W., McDonnel B., Elmqvist N.: Graphical perception of multiple time series. *IEEE TVCG 16*, 6 (2010). 3, 5

[KSB*09] Krzywinski M., Schein J., Birol İ., Connors J., Gascoyne R., Horsman D., Jones S. J., Marra M. A.: Circos: an information aesthetic for comparative genomics. *Genome research 19*, 9 (2009). 6

[LH14] Liu Z., Heer J.: The effects of interactive latency on exploratory visual analysis. *IEEE TVCG* (2014). 5

[LJH13] Liu Z., Jiang B., Heer J.: imMens: Real-time visual querying of big data. In *Computer Graphics Forum* (2013), vol. 32, Wiley Online Library. 6

[Mic] Microsoft: Displaying Graphical Execution Plans (SQL Server Management Studio). http://technet.microsoft.com/en-us/library/ms178071(v=sql.105).aspx. 2

[MW93] MacKenzie I. S., Ware C.: Lag as a determinant of human performance in interactive systems. In *Proc. INTERACT'93 and CHI'93* (1993), ACM. 5

[NAW*96] Nagel W. E., Arnold A., Weber M., Hoppe H.-C., Solchenbach K.: VAMPIR: Visualization and analysis of MPI resources. 2

[RH05] Reddy N., Haritsa J. R.: Analyzing plan diagrams of database query optimizers. In *Proc. VLDB* (2005). 1, 4

[SBB*10] Sigelman B. H., Barroso L. A., Burrows M., Stephenson P., Plakal M., Beaver D., Jaspan S., Shanbhag C.: *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep., Google, Inc., 2010. URL: http://research.google.com/archive/papers/dapper-2010-1.pdf. 2

[Shn96] Shneiderman B.: The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. IEEE Visual Languages* (1996). 2

[Sto86] Stonebraker M.: The case for shared nothing. *IEEE Database Eng. Bull. 9*, 1 (1986). 2, 3

[SWBW14] Simitsis A., Wilkinson K., Blais J., Walsh J.: VQA: vertica query analyzer. In *Proc. ACM SIGMOD* (2014). 2

[TSJ*09] Thusoo A., Sarma J. S., Jain N., Shao Z., Chakka P., Anthony S., Liu H., Wyckoff P., Murthy R.: Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB* (2009). 2

[Twi13] Twitter: Ambrose, 2013. http://github.com/twitter/ambrose/. 3, 6

[XKZC08] Xu Y., Kostamaa P., Zhou X., Chen L.: Handling data skew in parallel joins in shared-nothing systems. In *Proc. ACM SIGMOD* (2008), ACM. 4

[Zak11] Zakai A.: Emscripten: an LLVM-to-JavaScript compiler. In *Proc. ACM Object oriented programming systems languages and applications* (2011). 7