Davos: A System for Interactive Data-Driven Decision Making

Zeyuan Shang
Einblick Analytics
Cambridge, MA
zs@einblick.ai

Emanuel Zgraggen
Einblick Analytics
Cambridge, MA
ez@einblick.ai

Benedetto Buratti
Einblick Analytics
Cambridge, MA
bb@einblick.ai

Philipp Eichmann
Einblick Analytics
Cambridge, MA
pe@einblick.ai

Navid Karimeddiny
Einblick Analytics
Cambridge, MA
nk@einblick.ai

Charlie Meyer
Einblick Analytics
Cambridge, MA
cm@einblick.ai

Wesley Runnels
Einblick Analytics
Cambridge, MA
wr@einblick.ai

Tim Kraska
Einblick Analytics
Massachusetts Institute of Technology
tk@einblick.ai

ABSTRACT

Recently, a new horizon in data analytics, prescriptive analytics, is becoming more and more important to make data-driven decisions. As opposed to the progress of democratizing data acquisition and access, making data-driven decisions remains a significant challenge for people without technical expertise. In this regard, existing tools for data analytics which were designed decades ago still present a high bar for domain experts, and removing this bar requires a fundamental rethinking of both interface and backend.

At Einblick, an MIT/Brown spin-off based on the Northstar project, we have been building the next generation analytics tool in the last few years. To overcome the shortcomings of existing processing engines, we propose *Davos*, Einblick's novel backend. *Davos* combines aspects of progressive computation, approximate query processing and sampling, with a specific focus on supporting user-defined operations. Moreover, *Davos* optimizes multi-tenant scenarios to promote collaboration. Both empirical evaluation and user study verify that *Davos* can greatly empower data analytics for new needs.

PVLDB Reference Format:
Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Philipp Eichmann, Navid Karimeddiny, Charlie Meyer, Wesley Runnels, and Tim Kraska. Davos: A System for Interactive Data-Driven Decision Making. PVLDB, 14(12): 2893 - 2905, 2021.
doi:10.14778/3476311.3476370

1 INTRODUCTION

Over the last two decades the analytics space has drastically changed. Data has gone from scarce to superabundant and the popularity of data analytics as a means of making better decisions has exploded. However, taking advantage of data is hard as it requires technical expertise in data management, visualization, machine learning, and statistics, among other disciplines. This poses a significant challenge to decision makers, who usually have a deep understanding of the domain and problem, but not necessarily the technical skills to analyze all the available data. Despite the abundance of tools trying to make data analytics easier, existing tools still severely restrict domain experts from making data driven decisions on their own. Surprisingly, fully empowering domain experts to make data-driven decisions requires rethinking not only the interface of analytics tools but also the entire backend. This is rooted in the facts that (1) analytic interfaces are still based on two decades-old concepts, dashboards and workflow engines, and that (2) current backends do not aim to run complex often custom operations at the speed-of-thought.

As part of Einblick, an MIT/Brown spin-off based on the Northstar research project, we are aiming to build the next generation analytics tool for data-driven decision making. A key component of Einblick is its novel user interface for interactive data science, which won the VLDB demo award in 2015 [17] (see [7] for a more recent demo video). Unfortunately, we found that existing data processing engines and database systems are insufficient to fully realize the collaborative and interactive environment Einblick is aiming for. Existing systems were not designed to provide the interactive response times to enable live collaboration on data involving complex analytics functions. Thus, we started to design *Davos*, Einblick's novel backend, which combines aspects of progressive query processing, approximate query processing, and sampling. Moreover, Davos focuses on running complex operations and user-defined objects rather than standard SQL. Without Davos it would be impossible to power Einblick's user interface.

In this paper, we first motivate why there is a need for a new analytics tool and why existing data backends fail to provide a true collaboration environment for data science and data-driven decision making. We then outline the overall Einblick architecture and the design principle behind Einblick's novel progressive computation engine. To summarize, we have made the following contributions:

(1) We outline the shortcomings of existing tools for data-driven decision making.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476370

(2) We present a novel system *Davos*, which works in a fully progressive way by combining the data stream model, approximate query processing and sampling, with a specific focus on supporting user-defined operations.
(3) The system has a latency-aware resource allocation framework, including scheduling, storage management and caching to reduce the overall latency over multiple queries. We have been co-designing the system with the frontend since day one and propose various frontend-aware optimizations to further improve user experience.
(4) Both empirical evaluation and user study verify that *Davos* is able to greatly empower data analytics for new needs.

2 BACKGROUND

Current analytics tools can be roughly categorized into two groups[1]: descriptive analytics and predictive analytics tools.

2.1 Descriptive: "What happened?"

Descriptive analytics tools, like Tableau, Qlik, Thoughtspot, etc. help domain experts understand past data and are arguably the foundation of any business. They are used to examine how sales have developed and track manufacturing costs and numerous other factors in the past. Traditionally, descriptive analytics was done through reports, and the tools to create those were cumbersome, requiring extensive knowledge of SQL. The advent of Business Intelligence (BI) tools made it easier to understand past data. Broadly speaking, this change came over three generations of BI tools. The first generation moved users away from reports toward (interactive) dashboards and easy-to-use visual editors. The second generation lowered the barriers to entry even further by moving the software from on-premise applications, that were hard to install, to the cloud. The still-evolving third generation of BI tools, sometimes referred to as augmented analytics, aims to increase the self-service aspect of descriptive analytics by allowing users to ask "what happened"-type questions using natural language, among other things.

Over these three generations, BI tools grew in power and functionality, but their goal largely remained the same: create the best visualization of past data. Moreover, the manner in which users interact with BI tools has not changed much either. A single user creates a single visualization using various dialogues over a single dataset, then composes several of these visualizations into a dashboard so that others can view it.

However, before the user creates a visualization, data integration and cleaning are usually done with external tools, which sometimes come bundled with the BI tool. Unfortunately, this separation of cleaning and integration makes it hard to understand the underlying assumptions behind a visualization. While this can cause serious problems in some scenarios, this is not the case when dashboards are carefully curated by an expert for consumption by others.

2.2 Predictive: "What might happen?"

While understanding what happened is key to any business, it is a backward-looking approach. Often of equal interest is the question, "What might happen?", also known as *predictive analytics*. Here, machine learning (ML) and forecasting models are dominant. These technologies used to be the exclusive domain of highly trained statisticians or data scientists. More recently, tools like Alteryx, Data Robot, KNIME, etc. seek to make predictive analytics more widely accessible. These tools, sometimes referred to as self-service ML or Data Science platforms, provide visual user interfaces for building models and/or creating entire machine learning pipelines.

[1]Note, that this classification is based on the terminology introduced in [6]

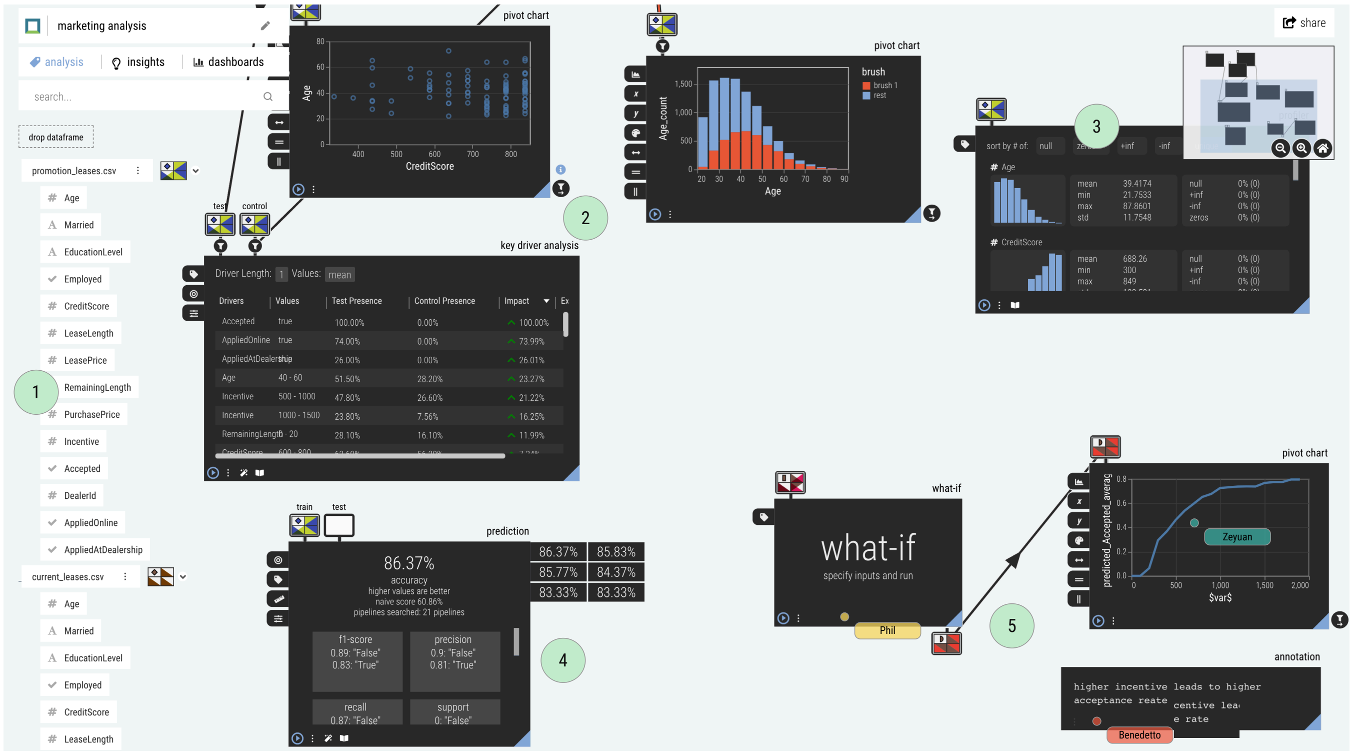
Interestingly, the user interfaces of self-service ML/Data Science tools are often quite different from the BI tools as they aim to create the best possible model for a given scenario. Instead of than dialogue-based interfaces, they are usually built on top of visual workflow engines, where individual operations are represented by boxes, which are then connected by the user to form an entire ML pipeline. The advantage of this interface is that it makes it easier to understand how the data "flows" from its source and raw format to the final model to eventually create a prediction. This is particularly important for ML as different ways of cleaning and encoding data can have profound impacts on the final accuracy of the model.

The downside of workflow engines, however, is that they do not provide any immediate feedback or interactivity. The user has to press a "play" button after curating the pipeline, which then starts the computation of the composed workflow, and it might take hours until the first result is produced. While some tools try to overcome this issue by providing more immediate feedback for parts of the pipeline through specialized interfaces (e.g., for hyperparameter tuning), they do so at the cost of ensuring that the user still sees and understands the whole process.

2.3 Towards prescriptive: "What should we do?"

While the focus of existing tools is on "What happened?" and "What might happen?", the real underlying question every organization wants to answer is "What should we do?". There are cases where the right action can be found just by understanding the past (descriptive analytics). For example, if the task is identifying the most underperforming sales person, the right action might simply be to fire them. However, in other cases, finding the right action might require building a forecasting model (predictive analytics). For example, a model can help to weigh the deals a salesperson might bring in over the next year. Other cases may require evaluating several scenarios and considering factors like the risk the salesperson might take some clients with them. Tools which help in these situations include what-if analysis, optimization tools like constraint solvers, and other techniques. Unfortunately, none of these techniques — which were originally framed as prescriptive analytics — are easy to use, and thus far can only be found in highly specialized verticals.

We believe that making the best decision cannot rely on what-if analysis or linear solvers alone. Rather, **finding the right action requires understanding the past (descriptive analytics), building models for the present and future (predictive analytics), and techniques to analyze different scenarios to optimize decisions (prescriptive analytics). It requires moving quickly between these modalities, as all three are needed to derive the best outcome.** At the same time, good decisions generally require neither pixel-perfect dashboards nor the best possible models. Instead, they need to be made in a timely fashion, and are



Driver Length: 1 Values: mean

Drivers	Values	Test Presence	Control Presence	Impact	Ex
Accepted	true	100.00%	0.00%	100.00%	
AppliedOnline	true	74.00%	0.00%	73.99%	
AppliedAtDealership	true	26.00%	0.00%	26.01%	
Age	40 - 60	51.50%	28.20%	23.27%	
Incentive	500 - 1000	47.80%	26.60%	21.22%	
Incentive	1000 - 1500	23.80%	7.56%	16.25%	
RemainingLength	0 - 20	28.10%	16.10%	11.99%	
CreditScore	600 - 800	62.60%	56.20%	7.24%	

sort by # of: null zeros +inf -inf unique

Age

mean	39.4174	null	0% (0)
min	21.7533	+inf	0% (0)
max	87.8601	-inf	0% (0)
std	11.7548	zeros	0% (0)

CreditScore

mean	688.26	null	0% (0)
min	300	+inf	0% (0)
max	849	-inf	0% (0)

86.37%
accuracy
higher values are better
naive score 60.86%
pipelines searched: 21 pipelines

f1-score 0.89: "False" 0.83: "True"	precision 0.9: "False" 0.81: "True"
recall 0.87: "False"	support 0: "False"

86.37%	85.83%
85.77%	84.37%
83.33%	83.33%

higher incentive leads to higher acceptance reate

Figure 1: Three users collaborate on analyzing marketing campaign data in an Einblick workspace performing various steps: (1) importing data from different sources, (2) descriptive analytics through visualizations and statistical methods, (3) profiling of a dataset, (4) automated ML prediction for marketing outcomes, (5) what-if analysis based on ML pipeline of various scenarios.

often only made once. The focus should therefore be on fast prototyping and making sure that everyone understands and signs off on the entire decision process, from data integration and cleaning, to modeling, to the final decision optimization.

2.4 Why are current data engines insufficient for prescriptive analytics?

Einblick aims to provide the first platform which combines, in one intuitive interface, visualization UIs as used for descriptive analytics, with workflow UIs as used for predictive analytics. Figure 1 shows a screenshot of the UI. Our vision is that every operation in Einblick immediately provides a visual response, regardless of the complexity of the operation or data size. Only that way are true collaboration and quick data-driven decision making possible. Finally, users should have access to the power of prescriptive analytics techniques, like what-if analysis, in a visual manner with the same interactive response times. Obviously, this poses a unique set of challenges on the data processing engine.

Interactivity According to a study by Liu [26], if latencies are greater than 500ms, user performance will degrade significantly. However, traditional analytical DBMSs, which are used widely as the backbone for descriptive and predictive tools, such as SAP HANA [19] or MonetDB [14], often take seconds, or even minutes, when computing results on increasingly large databases. Moreover, database like MonetDB were not designed to run complex analytics tasks. In contrast, systems like Spark can run complex analytics, but even starting a single job on Spark can take seconds [16] and running complex jobs over large data can take minutes to hours, making it impossible to collaborate in real-time on data problems; nobody wants to spend minute after minute waiting for a result during a meeting.

Complex operation Since more data mining and machine learning techniques are being used as part of data-driven decision making, the workflows are no longer limited to online analytical processing (OLAP). Instead, they are becoming much more heterogeneous. For example, almost all of our current customers frequently move between simple data exploration, using common visualizations like histograms, to performing key-driver analysis (a form of automatic statistical testing for significant differences), frequent pattern mining, building forecasting models, and performing what-if analysis. This requires a fundamental rethinking of the computation model, in a way that is flexible enough to accommodate for different computation needs. Further, with the increasing popularity of Python notebooks, users sometimes would like to integrate their custom operations into the system, i.e., user-defined-operations (UDOs).

Multi-Tenancy Decisions are rarely made alone, and thus with Einblick we made collaboration a first-class citizen. Most importantly, Einblick supports Google-doc like sharing of workspaces with real-time updates. That is, not only is the platform multi-tenancy, but one data analytics session might have several users working on it. This in turn poses a new set of resource management challenges including allocating both the computation (i.e., CPU cores) and storage (e.g., memory and disk) resources across multiple queries to maximize the overall user experience.

3 OVERVIEW

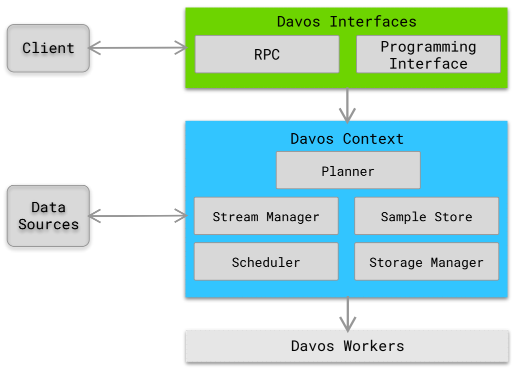


Figure 2: *Davos* Architecture: (1) Interface for submitting queries and receiving responses; (2) Context for generating and scheduling execution plans and managing streams, samples and storage; (3) Workers for executing the workloads.

Our goal with *Davos* is to build a data processing engine that addresses the above outlined challenges. We provide a brief overview of the system and describe individual pieces in later sections. Figure 2 shows *Davos*'s overall architecture.

The *Davos* context serves as the infrastructure for the whole system. It includes the stream manager which keeps track of the data streams such that the operators can communicate with one another. The sample store manages the data samples, e.g., which data source a sample is built from, such that if another query reads from the same data source, we can reuse the built samples. We discuss the management of samples and streams in Section 4. The planner optimizes queries and generates execution plans which we discuss in Section 5.4 and Section 5.5. The scheduler is in charge of scheduling jobs to the workers, which is covered in Section 6. The storage manager is responsible for the allocation and monitoring of memory and disk, and manages the cache for intermediate results as well, which we describe in Section 7.

Davos queries are Directed Acyclic Graphs (DAG) of data sources and steps, where each step specifies its operator (e.g., group, filter) and its inputs (e.g., from a data source or from a step). Such queries can be submitted to *Davos* through either Remote Procedure Calls (RPC, for frontend) or a Programming API (for debugging and development). Figure 3 shows an example query.

4 DATA STREAM

Data stream or dataflow is a natural solution for progressive computation, that is, all operators in the system consume and produce a sequence of data. The system as a whole sends a stream of responses back to the users, and the users can get the initial results quickly and receive incremental updates.

Davos further enriches this by introducing the concept of version, i.e., a data stream can have multiple versions. Each version is a complete result and a later version usually means better quality. Therefore, we can achieve better progressiveness in two dimensions: (1) the with-in-version progress; (2) the across-version progress. By having this multi-version semantics, *Davos* achieves a more fine-grained execution and response delivery mechanism to improve the progressiveness, without affecting the flexibility nor complicating the implementation, which we will show later. In later sections, we will use stream and data stream interchangeably.

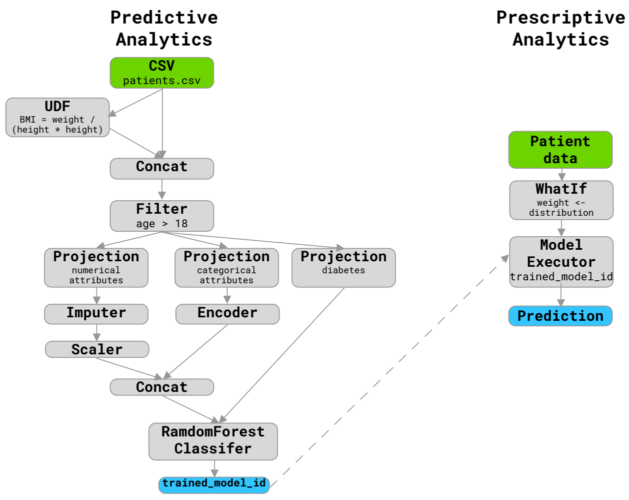


Figure 3: Two example queries (as DAGs): (1) Left shows a query for training a classifier predicting diabetes with complicated feature engineering; (2) Right shows a example of prescriptive analytics where we use a what if operator to understand how BMI affects predictions.

4.1 Data Abstraction

In *Davos*, each version of a data stream is a sequence of *Record Batch*, which is a collection of equal-length *Column Batch*es. Each column batch is an array of consecutive values for a single column, where we use the data layout of Apache Arrow [1]. It is a language-agnostic columnar memory format for flat and hierarchical data, organized for efficient analytic operations on modern hardware like CPUs and GPUs. Both record and column batch are *immutable*, i.e., we create new objects when modifying them. In later sections, we will use batch and record batch interchangeably.

4.2 Data Stream Interface

Our data stream interface exposes the following statuses and supports both blocking and non-blocking calls to retrieve the next *Record Batch* or the next version of the stream.

- ***HasNext***, there are one or more record batches ready and we can use *Next* to fetch the next record batch;
- ***Finished***, the data stream has finished and there will be no new versions;
- ***Deprecated***, the current version has been deprecated and we can use *NextVersion* to switch to the next version;
- ***Stopped***, the producer has stopped, thus the stream has been stopped and there will be no new batches;

- ***Failed***, the producer has failed, thus the stream is failed and there will be no new batches;
- ***Blocking***, the status is undetermined, and this is used by the non-blocking call of *Status*.

4.3 Data Sample

A data sample is essentially a data stream with extra metadata information, e.g., sample size, data source information, therefore it can be reused for queries reading the same data source. In Figure 2, *SampleStore* manages all data samples, including supporting querying samples, keeping tracking of sample memory/disk usage and evicting expired samples. Each sample has a time-to-live (TTL) and it will be removed once its TTL hits zero to keep its freshness. Users can also forcefully refresh a data sample for latest updates.

4.4 Publish-Subscribe Pattern

We adopt the publish-subscribe pattern for streams, that is, a producer (e.g., an operator) can publish a data stream and write batches to it, the consumers can subscribe to this stream and read batches out of it. By default the published data are not persistent, i.e., they are sent to the active subscribers when being published and the subscribers can only get data published after the subscription. Data streams can be marked explicitly as persistent such that the late subscribers can read the full history of published data, which is useful for reusing and caching. For example, when we create a sample for an expensive data source (e.g., executing a complicated query with many joins in a database), we can persist the data stream of this sample such that queries reading the same data source can reuse this sample to avoid reading the data source again. For now, all the persistent data streams are serialized to the disk and deserialized when needed.

4.5 Use Case: Progressive Histogram

We use histogram as a use case to justify the flexibility of progressive data stream semantic. Assume we want to reading from a large CSV file and compute the average of a column:

First response As soon as we read the first record batch out of the file, we will immediately publish it such that the aggregation operator can work on this batch and return the first response as fast as possible. Likewise for following record batches, therefore users can get progressive updates.

Sampling The multi-version semantic allows us to adopt such a complicated sampling strategy: while reading a file, we can take the first K rows and publish it as the first version, and at the same time maintain a reservoir sample and publish it periodically as newer versions. For example, whenever we have read every 10% of the full data, we can publish the current reservoir sample as a newer version. Subscribers will switch to the newer version once they find out the status has been updated to *Deprecated*. This hybrid sampling strategy provides a fast first response, while also providing better-quality responses (since they are built from the reservoir samples built over larger data) over time.

Final response While providing the results computed over the samples, we can launch another query over the full data (which is also running progressively over batches), and this ground truth result can be the final response in case that users want an fully-accurate result.

By combining the multi-version progressive data stream semantic and sampling techniques, *Davos* is able to provide better progressiveness while not sacrificing accuracy. The actual sampling strategy in *Davos* is more complicated. Please refer to Section 7.4 for more details.

5 EXECUTION

In this section, we first discuss the basic execution unit, *operator*. We support two types of operators: User Defined Function (UDF) and User Defined Aggregation (UDA). Further, we introduce the basic management unit, *job*. Lastly, we discuss a hybrid execution scheme to achieve a good trade-off between performance and flexibility, and how *Davos* rewrites a query plan to utilize a data store's internal processing capabilities.

5.1 Operator

Davos executes a query by creating several execution units based on the query and these execution units communicate with each other through data streams, i.e., to read the outputs of an execution unit, another execution unit can subscribe to its output data stream. In *Davos*, we call these execution units *operator*s.

Since data arrives as a sequence, all operators run in the same manner, that is, an operator implements a processing function taking record batches as inputs and producing record batches as outputs. For example, for the filter operator, this function takes in a record batch and produces a record batch with rows selected by the filter. Another example is that for the horizontal concatenation operator, its function takes into multiple record batches and concatenates them horizontally (i.e., merging all columns) as the output record batch.

Besides the function for processing batches, an operator might want to take different actions when the status of input data streams changes. For instance, for the aggregation operator, when the input stream becomes deprecated, it has to reset all the aggregated results, e.g., setting the accumulated sum as zero. To support this, *Davos* provides multiple trigger functions (e.g., on inputs being deprecated, on inputs being failed, on operator being stopped, etc) such that operators can implement their own logic.

In this sense, *Davos* is a push-based query engine. When an operator's input stream's status have been determined (i.e., not in the status of *Blocking*), it will check the status (and record batches if applicable) and apply the processing or trigger function accordingly. This routine is the basic scheduling unit in *Davos*, which will be covered in Section 6.

5.2 UDF & UDA

To ease the implementation of operators, *Davos* offers two more detailed abstractions of operators:

- **User Defined Function (UDF)**, which returns the stateless-transformed results over the input batches, e.g., project, filter in databases;
- **User Defined Aggregation (UDA)**, which returns the stateful-aggregated results over the input batches, e.g., aggregation in databases.

For example, for a sequence of batches (*A*, *B*, *C*), a UDF outputs (*func(A)*, *func(B)*, *func(C)*), whereas a UDA outputs (*func(A)*, *func(A+B)*, *func(A+B+C)*), where *A+B* is the vertical concatenation of batch *A* and *B*.

Whether an operator is a UDF or UDA has a direct impact on the version of its output data stream. For a UDF, it simply publishes a output batch to the current version of the output stream. Instead a UDA has to deprecate the current version of its output stream and publishes the output batch to the new version, such that the downstream operators will take actions (e.g., resetting its states) accordingly. A UDA can be *non-incremental* or *incremental*, depending on whether they need to see all previous batches, for example, computing the sum is incremental, while training a ML model can be non-incremental.

UDFs or UDAs implement the following interface whose functions are executed at different moments:

- **Open**, which is executed when being initialized;
- **Reset**, which is executed every time after the input streams have been initialized or deprecated;
- **Close**, which is executed when being finalized;
- **Process**, which is executed every time when the inputs become ready for processing.

5.3 Job

A job, as a DAG of operators, is analogous to the access plan for a query in databases. It manages and coordinates the operators, for instance, we schedule the execution (in the scheduler) and manage the memory usage (in the memory manager) for each job.

To execute a job (i.e., computing the next result), we adopt a bottom-up approach to traverse the DAG. That is, for an operator, if its inputs are ready then we can run *Process* to produce outputs and then go to its downstream operators (i.e., operators subscribing to it), otherwise we go to the upstream operators and repeat the procedure. By starting from the last operator, we are essentially following the critical path to compute the next result.

5.4 Hybrid Execution

Davos supports executing operators written in C++ or Python. In general, basic analytical operators (e.g., project, filter, aggregation) are implemented in C++ and machine learning or data mining operators are implemented in Python (e.g., random forest classifier). The decoupling of operators through data streams makes this possible, and Apache Arrow's zero-copy data access across languages makes the serialization/deserialization cost negligible.

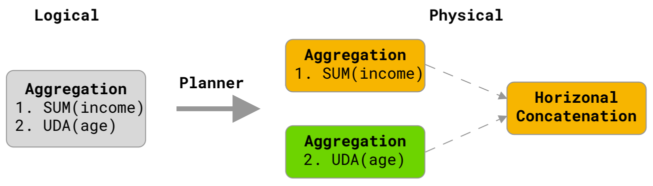


Figure 4: Hybrid execution: yellow operators are native C++ operators and green operators are running in Python runtime. The dashed lines indicate that these operators don't communicate directly instead through data streams.

This hybrid scheme extends the functionality of *Davos* compared with other engines, as well as allowing for more fine-grained execution to promote both efficiency and flexibility. For example, for aggregation, we have some predefined aggregation methods implemented in C++ (e.g., sum, count, average or so) and we can also let the users provide their custom aggregation method written in Python (as a UDA). This is done by leveraging query rewrite and planning as in Figure 4. Given the description of the aggregation, *Davos* extracts the predefined aggregation methods and creates a new native operator for them and likewise it extracts the user-defined aggregation methods and creates a user-defined operator. The system further adds a horizontal concatenation operator to merge the outputs of this pair of aggregation operators.

5.5 Pushdown

For certain data sources (e.g., DBMS), *Davos* can restructure a query plan to utilize the internal processing capabilities of the underlying data store. For example, we support push down sampling, predicate and join to DBMS to leverage its efficient implementations of indexes and algorithms.

6 SCHEDULING

We have discussed the execution scheme for a single job in the previous section. Moreover, for a multi-tenant use case, it is crucial to schedule jobs across multiple users. In this section, we will formulate the scheduling framework in *Davos* and then propose the adaptive scheduling strategy.

6.1 Framework

A scheduling *task* for interactive data systems should be small (in terms of workloads) to allow for fine-grained control and fast responses. On the other hand, if the tasks are too small, there will be too many tasks for a single job, incurring overhead in scheduling. As discussed in Section 5.1, we scope a scheduling task as a routine function that pulls the input stream and applies a processing or trigger function. For example, for filter, the routine pulls the input stream and gets one record batch (if the status is *HasNext*), computes a filtered batch and publishes it to the output stream. If the input stream's status is not *HasNext* (e.g., *Failed*), this routine applies other trigger functions accordingly (e.g., *OnFailed*). The scheduling framework assigns pending operators to workers in a pool which in turn execute the operator's routine function as a task.

6.2 Strategy

The adaptive scheduling strategy is priority-based. Ideally, the priority of a job should have a strong correlation with the overall user experience, although the latter is hard to quantify, we believe the following factors have a strong influence: (1) the first response time, in that a user would like to get the first response as quickly as possible; (2) the quality of responses over time, in that a user would like to see the quality of response improve as much as possible. Based on this, our scheduling strategy gives the highest priority to a new job (i.e., a job without any computed results yet) and prefers jobs with low quality.

Priority Considering the variety of operator semantics, it is difficult to measure the quality of a job by examining its outputs without operator-specific logic. Instead, we use the square root of the average number of processed input rows as the approximation for the overall quality of a job, that is, $Q = \sqrt{R}$. This is based on the observation that the error of many operators (e.g., aggregation) is proportional to the inverse of the square root of the input size. Between each result of the job, we have the quality change ΔQ and the elapsed time ΔT, and $\frac{\Delta Q}{\Delta T}$ approximates the relative quality improvement for scheduling this job. However, if the job has made good enough quality, we should dis-prioritize it to favor early-stage jobs, therefore we further normalize the above value by dividing Q.

Besides, there are different types of jobs and we want to prioritize across types, for example, we might want to prefer analytical jobs instead of dumping jobs (e.g., dumping the outputs to a CSV file), because users usually have less patience for analytical results. As another example, a job in the backend corresponds to a visualization in the frontend, if the visualization is moved out of the screen by the user, we shall dis-prioritize its corresponding job. To this end, we define a weight W to adjust the priority based on job type and visual status, e.g., a large weight for some jobs expected to have very high priority. In summary, the priority of a job is now $\frac{\Delta Q \cdot W}{\Delta T \cdot Q}$.

Task extraction By leveraging the adaptive scheduling strategy, we are able to find the job with the highest priority. To extract a task from job, we want to find the most useful task towards computing the next result. To achieve this, we traverse its DAG by a bottom-up approach (i.e., starting from the last operator, for example, the aggregation operator in Figure 3) and find an idle bottleneck" operator, whose inputs are ready and we can run its *Process* to produce outputs. By starting from the last operator, we make sure that we find the most pressing task for next result.

Implementation Similar with the BF scheduler[5], *Davos* has a global job queue. To improve the concurrency, we employ a *copy-on-write* queue, that is, each worker gets an immutable snapshot. Furthermore, the priority of a job changes only when there is a new result, therefore we store the priority with the job and let the worker computing the next result update it, while other workers can just use the stored value to avoid redundant computation.

7 STORAGE

In this section, we discuss how we deal with data storage in *Davos*. We first introduce the data model and illustrate the memory and disk management mechanism, including streams, runtime and samples. Lastly, we discuss the caching strategy.

7.1 Data Model

As discussed in Section 4.1, *Davos* uses a columnar format for the data representation and adopts the underlying layout of Apache Arrow [1], where data are stored as consecutive arrays. Arrow leverages a memory pool to allocate the memory space and keep track of the memory usage. Although this is a decent solution for memory management, it requires that all data stay in the memory and prevents us from using disk as a secondary storage. Therefore, we add another layer of abstraction between the data abstraction (i.e., record batch and column batch) and the actual Arrow array data, that is, the *array wrapper*. Figure 5 gives an example.

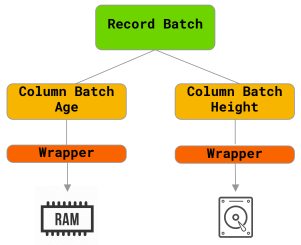


Figure 5: Array wrapper: a record batch has two columns and with the wrapper layer, one column can be stored in memory and the other one can be stored on disk.

To dump a in-memory array onto the disk to free some memory space, we can call *Serialize()*. Conversely, to read the data out, we can use *data()* to get the in-memory Arrow array, which reads the data from the disk, deserialize it (only if the array has been dumped before) and puts it back into memory. Besides, we have memory and disk usage accounting, which are used for memory and disk management. By having the wrapper layer, we achieve the flexibility for data storage while keeping data access transparent to the column and record batch.

It is straightforward to estimate memory or disk usage for a column or record batch. Next, We explain how to estimate the memory usage for a data stream and the estimation for disk usage can be done likewise.

Data stream memory usage estimation Since we adopt the publish-subscribe pattern for data streams, we can use publishers to estimate the memory usage. That is, for a publisher, if it is a persistent data stream (i.e., all the record batches are persistent in memory or on disk), we just sum up the total memory usage of all batches. Otherwise, because subscribers read data in the same order as they are published, and the underlying data are shared between subscribers, we only need to consider the maximum memory usage among all subscribers (i.e., the memory usage of the *slowest* subscriber). The memory usage of a subscriber is simply the sum of memory usages of all batches that are to be fetched.

7.2 Memory Management

Assume that a machine has a total memory capacity of X, and we reserve $M\%$ of it as the memory space of *Davos*. In practice, we set $M = 90$. There are three major usages of memory in *Davos*: (1) *Data Sample* for storing the samples of loaded data sources (i.e., persistent data streams), for which we allocate $S\%$ of the total memory; (2) *Data Stream* for storing all data streams for the communications between operators, for which we allocate $D\%$; (3) *Runtime* for the memory used by executing the operators (e.g., in-memory data structures, ML model), for which we allocate $R\%$. Based on our own experiences while tuning the system, we find that $(S, D, R) = (30, 30, 30)$ is a reasonable configuration (where $S + D + R = M$). We further discuss how to estimate and control the memory usage for these purposes.

Data sample Since a data sample is merely a data stream plus some metadata information, the memory usage of a data sample is simply the sum. We discuss how to control both memory and disk usage for samples in Section 7.4.

Data stream The stream manager manages all publishers and subscribers, therefore it is straightforward to compute the total memory usage of all data streams. Since operators produce data to the data streams, we can limit the memory usage of data streams by controlling the scheduling of operators. That is, if an operator's output stream has consumed too much memory (which means its subscribers are relatively slow consumers), we shall not schedule it until its downstream operators have consumed the data. In practice, we set the threshold as *256MB*, considering that there can be hundreds of operators running in *Davos*.

As an example, assume we have an operator reading a Parquet file (i.e., reader) and another operator doing reservoir sampling from the former's outputs (i.e., sampler). Reservoir sampling is slower and it is possible that the reader produces too much data for the sampler to process. In this case, when the reader's output stream has more memory usage than the threshold, we will not schedule the reader until the sampler has processed the data.

Runtime There are both C++ and Python operators running in *Davos* simultaneously. To keep track of the memory usage of an operator, memory profiling doesn't always work without adding too much overheads, especially in C++. Therefore we leverage a self-reporting mechanism for the internal operators implemented by us. That is, an operator can override a method *GetMemoryUsageInBytes* to report its memory usage. For example, for a join operator, it can compute the memory usage of its index in the method. For external operators (i.e., user-defined ones), since they are Python-based, we use Python's memory profiling mechanism to estimate the memory usage. If the total runtime memory usage is beyond the threshold, we simply kill the job consuming the most memory.

7.3 Disk Management

The disk serves as the main storage for persisting data streams. Persistent data streams are used for two purposes: (1) *Data sample*, a data sample will always be persistent on disk and it will be deserialized into memory when being used, for which we allocate *S%* of the total disk space for *Davos*; (2) *Intermediate result*, that is, caching the results of some operators because they are expensive to compute, e.g., join and Python scripts, for which we allocate *C%*. We find that $(S, C) = (60, 40)$ is a good configuration based our experience using the system.

Data sample Similarly, the disk usage for a data sample is straightforward to compute. We discuss how to manage the memory and disk usage for data samples in Section 7.4.

Intermediate result We discuss the caching eviction strategy in Section 7.5.

7.4 Sampling

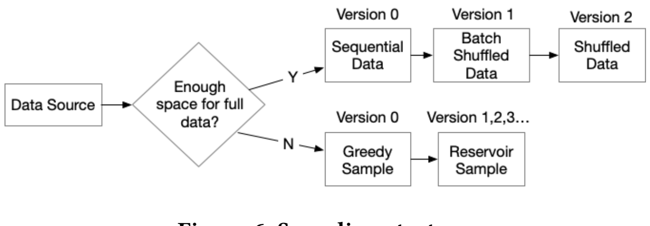


Figure 6: Sampling strategy

Algorithm 1: Sample space reservation

```
  Input: $S_{free}$, $S_{data}$
1 if $S_{data} \leq S_{free}$ then
2     return $S_{data}$
3 $s = \min(S_{min}, S_{data})$;
4 while $S_{free} < s$ do
5     Break if the largest sample size is no larger than $S_{min}$;
6     Shrink the largest samples to the second largest samples
        and update $S_{free}$;
7 if $S_{free} < s$ then
8     return Error("No free space!")
9 return $\min(S_{free}, S_{data})$;
```

Sampling strategy Figure 6 shows the sampling strategy. For a data source, if we can know its size upfront and it fully fits in the available sample space, we use the full data and create the first version of the sample by reading sequentially. Next, we use the batch-shuffled data as the next version, which only shuffles the order of batches (thus much faster). Finally we use the fully shuffled data as the final version.

Otherwise, either the data source's size is unknown or it cannot fit fully, we create a down-sized sample. First we return the first K rows as the first version, where K is decided by the sample space reserved, this is a greedy sample. We maintain a reservoir sample simultaneously and publish it as a newer version with a fixed interval, where in practice we do that after reading every 10% of data, such that we have multiple versions for the reservoir samples and later versions have better quality because they are built over larger data. In practice, we use different methods to compute the progress, for example, for databases, this is easily done by checking the row counts; for a file-based data source (e.g., CSV or Parquet), this can be estimated by comparing the position of file pointer and the total file size.

Sample space reservation Algorithm 1 presents the sample space reservation algorithm. First of all, given a data source, we need to determine the sample space we want to reserve. Assume that the current free sample space is S_{free} and the size of the data source is S_{data}, if we have enough space for the full data source (i.e., $S_{data} \leq S_{free}$), we just store the full data. Otherwise, we have a threshold for a reasonable sample size S_{min}, which is 1GB currently, and we use the lower value of these two as the minimum sample size s, which is the sample size we would like to reserve.

Our sample space reservation algorithm aims to treat each sample fairly and make samples have comparable spaces. To this end, we adopt the idea of diminishing return, that is, we keep shrinking the largest sample down to the second largest sample while keeping the sample size beyond S_{min}, until we have enough free space or we cannot shrink any more. If we still cannot find enough space, we will return an out of capacity error to let users know.

We use Algorithm 1 to reserve the disk space for samples. Since a sample is always stored on the disk, we employ the least recently used (LRU) strategy to evict samples in memory.

7.5 Caching

There are two aspects about caching, i.e., *what to cache* and *what to evict*. For the latter aspect, we simply use the LRU strategy. For the former one, we adopt an adaptive approach to provide hints. When a job is finished, we check the whole DAG and find the sub-graphs that fulfill the cost, frequency and space constraint: (1) *Cost* constraint is that this sub-graph must be costly (in terms of time) enough to compute and currently we take 5 seconds as the minimum threshold; (2) *Frequency* constraint is that this sub-graph must be used frequently recently (e.g., used 5 ore more times), and we maintain a frequency table and reset it periodically to keep its freshness; (3) *Space* constraint is that the sub-graph must not take too much space and currently we enforce that it cannot take more than 10% of the total space for caching.

After getting the hint, *Davos* will cache this sub-graph next time and later queries with the same sub-graph can reuse the cached intermediate results.

8 FRONTEND-AWARE OPTIMIZATIONS

Being co-designed with frontend since day one, *Davos* has several optimizations for improving the user experience.

Stopping mechanism One important observation is that users usually do a lot of operations back and forth in a short period of time at the frontend, i.e., the trial-and-error process. The frontend sends a stop query request to *Davos* when a query is no longer needed (e.g., the user changes the parameter or removes the operation). At the backend, we carefully design and implement a *fast stop* mechanism, that is, we have a signal variable for each job and the operators will periodically check it while executing and stop the execution as soon as possible when the stop signal is up. Besides, we assign a being-stopped job with the highest priority in scheduling, such that it can finish up the stopping (e.g., some clean ups) as soon as possible. By having such a fast stopping mechanism, we can save much more resources by not executing unnecessary jobs.

Skipping responses Considering the responses sent from *Davos* will be visualized in some way at the frontend and then perceived by the users, we can safely skip some responses when there are too many of them generated in a short period of time, since the users would not notice it. For example, when we have a response to send in *Davos*, if the previous response was sent 100 milliseconds ago, we can wait for another 100 milliseconds and see if there are newer responses produced. If there are, we can simply send the newest response and skip the responses in between, such that we can save more network bandwidth and also reduce the pressure of visualization at the frontend.

Utilizing frontend feedbacks To better improve the overall user satisfaction, the frontend can send some feedbacks to *Davos* to help it make better decisions in many aspects through user interactions. For example, the frontend can give *Davos* a hint for scheduling priority in the following scenario. When a user drags a ongoing operation out of the screen, the frontend can then let *Davos* know it should decrease its priority in scheduling to favor other on-screen jobs.

9 IMPLEMENTATIONS

We use C++ and Python extensively for building *Davos*, with roughly 19,000 lines in C++ and 9,000 lines in Python. In this section, we discuss several implementation highlights.

Secure UDF/UDA sandbox Users are able to implement their own UDF/UDA operators through the frontend, which we call *external* operators. Although it provides great flexibility for users, it brings up some potential security issues. Users might write malicious code either intentionally or unintentionally, or a operator might consume too much resources (e.g., disk or network) that are hard to keep track of and limit. We implement a secure sandbox mechanism by utilizing Docker and Arrow Flight (a RPC framework for exchanging Arrow data). When we initialize an external operator, we create a Docker container running the Flight server, and when the operator is scheduled to run a function (e.g, *Open*, *Process*), we send an action from the main process to the container through RPC, the container executes the action and sends the results back to the main process. Besides, the operator in the main process monitors the container statistics and kills the container when it overuses the predefined resources.

User-defined-expression Besides the external UDF/UDA operators, users can write one-line expression to create new columns or filter rows in Python syntax. For example, they can easily calculate Body Mass Index (BMI) with *weight / (height * height)* or get the first name of a customer with *FullName.split(' ')[0]*. We use the hybrid execution scheme again to achieve the trade-off between flexibility and efficiency, that is, during the query rewrite phase, if a expression can be efficiently executed in C++, we create a C++ operator for it, otherwise we use its Python counterpart. For C++, we implemented a parser for the C++-compatible expression (which is a subset of Python), which creates LLVM expressions using Gandiva, a LLVM-based expression compiler for Apache Arrow.

10 EVALUATION

We aim to answer the following questions: (1) By adopting the idea of progressive computation in many aspects, how much advantages we can achieve for different queries individually and for the whole workloads? (2) How the quality of query results change over time? (3) By optimizing for the multi-tenancy scenario, how much benefits in terms of the distribution of latencies we can acquire?

10.1 Experimental Setup

Workloads We formulate three workloads to evaluate *Davos*: (1) *Micro*, which is a set of representative queries we create for each operation at the frontend, e.g., histogram, machine learning and correlation. We demonstrate the flexibility, efficiency and progressiveness of *Davos* in a single-tenant scenario through this workload. (2) *Synthesized*, which we utilize *IDEBench* [18] to generate the workloads. More specifically, we use its data generator to scale up the datasets (with preserving the distributions and relationships between attributes) and its workload generator to generate a larger number of queries. Using this workload, we create some extreme cases to verify the scalability and robustness of *Davos* and evaluate the effectiveness of different design choices for the multi-tenant scenario. (3) *Real*, which we collect by having six people using the platform simultaneously with different tasks for an hour. With this workload, we are able to understand the system's performance in the real world.

Datasets We collected two real world datasets for *Micro* and *Synthesized*: (1) *MIMIC*, which is a medical dataset for patients, including their demographic information (e.g., age and gender) and medical records (e.g., blood glucose and the diagnosis of diabetes). (2) *Flight*, a dataset containing U.S. domestic flights from the *IDEBench* paper [18]. We utilized the data scaler tool in [18] to create a scale-up version for each datasets, including *MIMIC*-1000 and *Flight*-100. Table 1 describe the characteristics of these datasets. For *Real*, the users collected 27 datasets on their own, with size ranging from several KB to 959 MB and the average size is 40MB.

Table 1: Dataset Overview

Dataset	Size	Rows	Columns
MIMIC-1000	9.9 GB	33,827,000	71
Flight-100	27 GB	500,000,000	12

Measurements There are two aspects matter: (1) Latency, which includes the first response time for a quick initial result and the last response time for the final result; (2) Quality, which is the result quality over time. However, the definition of quality varies across different operations(e.g., histogram, machine learning). In this evaluation, we focus on the quality of the histogram operation (i.e., the accuracy of the aggregation result) and AutoML operation (i.e., the predictive power of the returned model).

Baselines Per the unique characteristics of *Davos*, there doesn't exist a similar system to compare with. To verify the effectiveness of progressive computation, we compare *Davos* with its blocking-execution variant *Blocking*, that is, running the computation over the full data to completion. For certain analytical workloads (i.e., histogram), we compare with MonetDB [9], a state-of-the-art open-source analytical column-oriented DBMS, and with Pandas [10], which is an open source data analysis and manipulation tool built on top of Python with great popularity in data science.

Hardware environment All experiments were conducted on a `c2-standard-16` node running on Google Cloud[8]. It has 16 3.10 GHz virtual CPUs and 64 GB RAM running 64-bit Ubuntu 18.04 with Linux kernel 5.4.0, GCC 7.5.0 and LLVM 10.0.

10.2 Progressive Computation

Single query response time Figure 7 describes the response times for different queries on *MIMIC-1000* and *Flight-100*, including the first and last response of *Davos*, and its blocking-execution variant. We have the following observations. (1) For all queries, *Davos* is able to deliver the first results within the first 10 seconds, which is much faster than *Blocking*; (2) For some queries, *Blocking* and *Davos* returns the last response at almost the same time, because those queries are incremental (i.e., aggregation) and don't incur too much overheads with progressive computation. (2) For some queries, *Blocking* returns a faster response than the last response of *Davos*, because for non-incremental queries, progressive computation triggers more executions thus more overheads; (3) For some queries, *Davos* returns a faster last response, because a query usually contains numerous operators, with proper progressive computation, the execution is pipelined and every operator is running in parallel with little idle time. While for the blocking execution, an operator has to wait for its upstream operators to complete before start executing.

Optimistic and pessimistic Based on the real world workload *Real*, we measure the benefits for the whole workload by adopting progressive computation. That is, we consider two cases for the waiting time between each query in the user interactions, that is, (1) *optimistic*, in which the user submits the next query after the first response of the previous query arrives; (2) *pessimistic*, in which the user submits the next query after the previous query completes. Figure 8 compares these two cases, and we find that the progressive computation streamlines the user interactions and thus greatly improves the interactivity of the system. Moreover, the performance gap increases over the data scale, which means progressive computation have more advantages for larger data.

10.3 Quality Analysis

A quick first result only matters with good quality. To understand how the quality of result changes over time, we pick two representative jobs: `histogram` and `automl`, because their qualities can be clear-defined and quantified. For `histogram`, we define the quality as one minus the relative error to the ground truth. For `automl`, it automatically finds a sequence of good models given a task description and train/test split, therefore we use the F1 score (for classification) or the minus mean-squared error (for regression) over the test dataset as the quality. Figure 9 shows the change of quality over time under different circumstances.

Quality: end-to-end Figure 9(a)(e) show the end-to-end case, where the dataset is not loaded in advance and the computation and data loading starts in parallel. The quality goes up over time as the system has processed more data and eventually achieves perfect quality. This exhibits a great advantage of progressive computation, that is, we get a quick initial result and results are refined over time, such that users are able to get a general idea of the results without waiting for the completion. On the other hand, it provides opportunities for early-stop if results are are already acceptable. Please note that for both datasets, we sort them by some attributes and let `histogram` aggregate over these attributes. This creates a much more "difficult" case for progressive computation because the system has to almost fully scan the data to get highly-accurate results for the sorted-attributes.

Quality: sampling Figure 9(b)(f) verify the effectiveness of sampling. Since *MIMIC-1000* can fully fit into the sample memory space, *Davos* creates two versions of full samples (sequential and shuffled). For *Flight-100*, because the full dataset is not able to be stored in memory (the sample memory space is around $0.3 * 64 = 19$ GB), *Davos* creates multiple versions of reservoir samples. As we can see, samples can greatly speed up the increase of quality compared to the original data, and later versions of samples can offer better quality because they are created based on larger data.

Quality: preloaded Figure 9(c)(g) illustrates the quality over time for preloaded datasets, i.e., *Davos* have created samples in advance. They are essentially a "zoomed-in" version of the last samples in Figure 9(b)(f). With samples, we are able to converge within 10

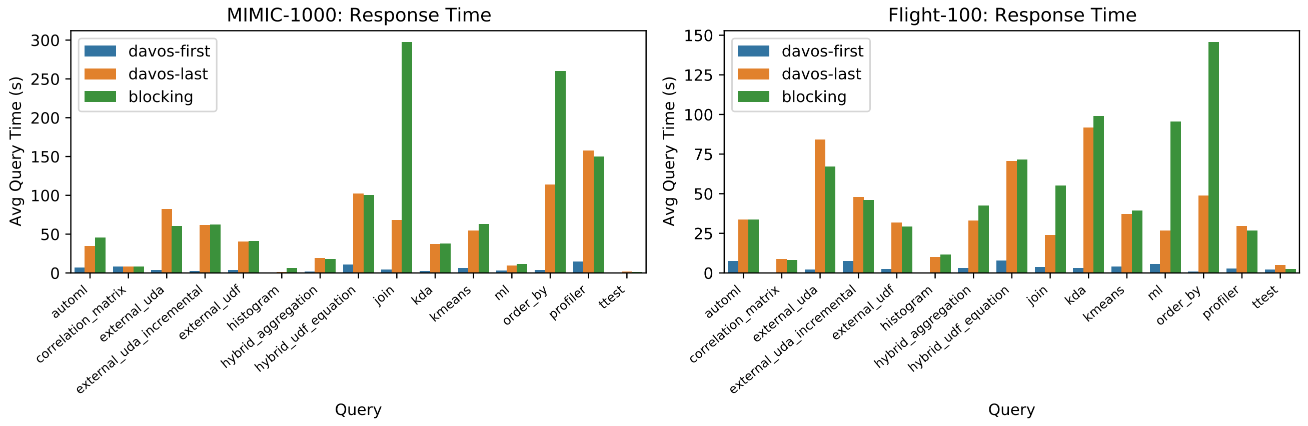


Figure 7: Response time: x-axis shows the type of queries. Note join is slower on *MIMIC-1000* because its selectivity is higher.

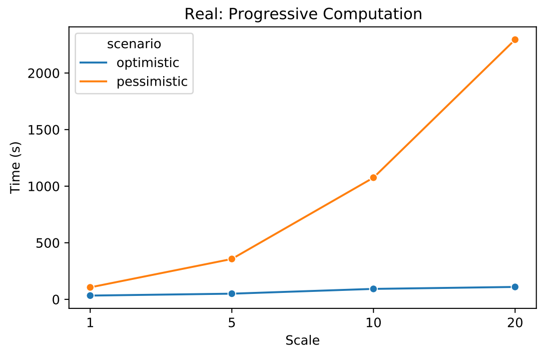


Figure 8: Comparison between *optimistic* and *pessimistic* computation: x-axis shows the scale of the datasets and y-axis shows the total execution time.

seconds for both datasets. For *Flight-100*, since it is a down-sized sample, *Davos* doesn't produce a perfect result, which might require more complicated sampling mechanisms like stratified sampling.

Quality: AutoML In Figure 9(d)(h), for `automl` queries, *Davos* generates the first better-than-naive solution within 10 seconds, followed by a sequence of better results within 30 seconds.

Comparison with *MonetDB* and *Pandas* For `histogram`, we compare with *MonetDB* to understand the performance difference between *Davos* and a state-of-the-art analytical database employing a blocking execution model such that users have to wait until an exact query result is computed. Simultaneously, we compare with *Pandas* since *Pandas* is widely used by data scientists for data analytics. Besides the execution time, we measure the time that it takes to load the dataset as the *loading time*.

(1) As we can see from Table 2, it takes several minutes for *MonetDB* and *Pandas* to load the whole dataset before starting any computation. Instead, *Davos* can immediately start the computation and stream progressive results back to the users while loading the dataset. (2) *MonetDB* is extraordinarily fast because it has been designed and optimized towards analytical queries However, if we compare it with *Davos* (assuming the data has been loaded), the gap is not huge. Considering *Davos* doesn't implement intra-operator parallelism (while *MonetDB* can use all the cores), the single-threaded performance of *Davos* is impressive. Further, *MonetDB*'s performance will degrade dramatically when the dataset doesn't fit in memory, as verified by *Flight-100*. (3) *Pandas* is the slowest for both datasets. It cannot handle *Flight-100* and we have to read into chunks. At the opposite, *Davos* automatically creates samples and if the users would like to get a accurate result over the full data, *Davos* is able to perform progressive computation without caching samples, which doesn't require full data to fit into memory.

Table 2: Performance of *MonetDB* and *Pandas*

System	Dataset	Loading Time (s)	Execution Time (s)
MonetDB	*MIMIC-1000*	243.04	2.973
Pandas	*MIMIC-1000*	177.06	14.53
Davos	*MIMIC-1000*	187.48	6.00
MonetDB	*Flight-100*	662.18	32.614
Pandas	*Flight-100*	366.00	146.93
Davos	*Flight-100*	295.86	10.25

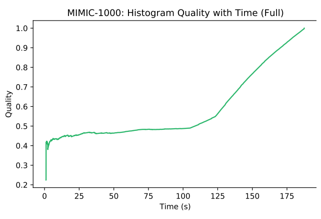
10.4 Multi-tenancy

We ran the multi-tenant experiments using *Synthesized*, which includes 120 analytical queries (e.g., filtering, sorting, aggregation and join), 20 AutoML queries and 20 user-defined queries. To simulate a user's interaction, we assume each interaction takes 60 seconds and a user might send 1 to 3 queries. We have 16 threads running the interactions at the same, where each thread runs 5 interactions sequentially, therefore the whole experiment runs for 5 minutes.

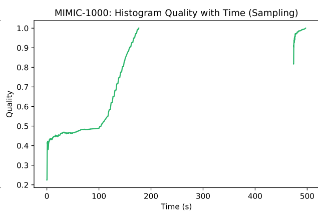
Figure 10 shows the distribution of response times for all queries. Each query gets its first result within half an minute, and for the analytical queries each query gets its first response within 1 second. More than 85% queries are done within the first 10 seconds.

10.5 User Study

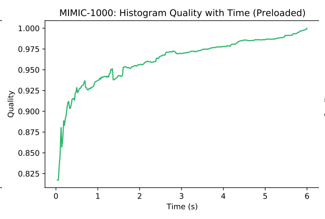
We empirically evaluated the benefits of using a progressive computation engine like Davos with a user study. We recruited four participants from the popular freelancing platform Upwork and



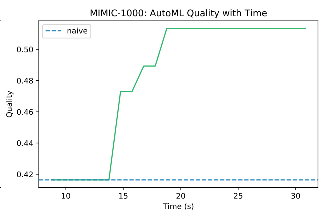
(a)



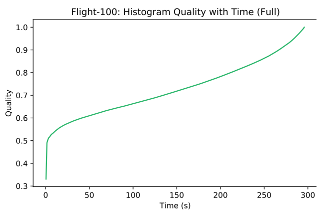
(b)



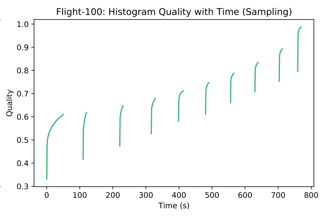
(c)



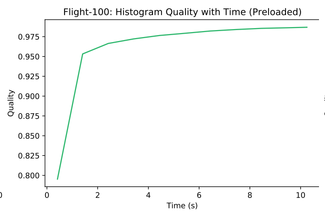
(d)



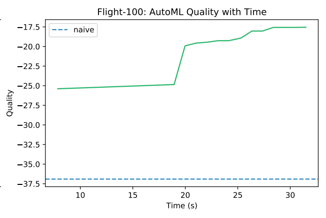
(e)



(f)

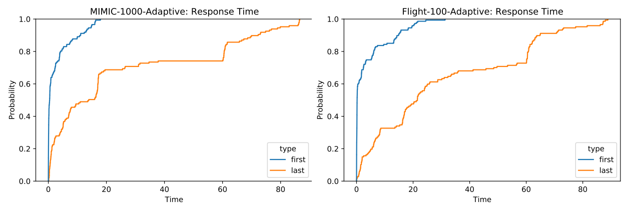


(g)



(h)

Figure 9: (a)(b)(c)(e)(f)(g): Quality over Time for Histogram, where the quality is measured by one minus the relative error to the ground truth. (d)(h): Quality over time for AutoML, where the quality is measured by the F1 score over the test split.



(a) (b)

Figure 10: Latency distribution for multi-tenant

asked them to solve the same task based on a Kaggle challenge, using either Einblick or Jupyter Notebook. All the participants had formal training in data science and programming and between one to five years of experience in the field.

We had univocal positive feedback by the Einblick users that reported, "it is really easy to use. After 3 hours you can do everything you want... fantastic" and "The application is fantastic. It allowed me to visualize and understand a dataset I knew nothing about.". For more details about the user study, please refer to our full technical report.

11 RELATED WORK

Interactive Data Exploration. There are many data systems for interactive analytics, e.g., Dremel [27], Drill [20], Presto[29] and Druid [30]. However, most of them only support OLAP analytics, while *Davos* is highly extensible for machine learning, data mining and user-defined workloads. Besides, they still adopt the traditional blocking query model (i.e., the user has to wait for the completion of the whole query), while the computation in *Davos* is progressive and the results are refined over time.

Streaming systems. Streaming systems, including Spark Streaming [31], Storm [4] and Flink [2], adopt a stateful computation model to enable incremental and continuous computation over data streams. But these streaming systems are not optimized for a multi-tenant scenario, e.g., they leave the scheduling across different jobs to some cluster-level generic resource management frameworks like Apache Hadoop YARN [3]. Further, despite they provide real-time response over continuous queries, they don't have a native support for data samples.

Approximate Query Processing. There have been two major categories for AQP techniques: (1) stratified or biased sampling [11] and (2) online aggregation [22]. For stratified sampling, most systems aim to build better samples based on the query workloads (e.g., AQUA [12], BlinkDB[13], DICE [25] and AQP++[28]). These systems typically require knowledge about future workloads or expensive preprocessing of samples, which goes against the ad hoc nature of interactive data exploration. Online aggregation systems (e.g., CONTROL [21], DBO [23], HOP [15], FluoDB [32] and SwiftTuna [24]) sample data incrementally produce a confidence interval of estimated results, and this interval converges as the query is computed progressively. *Davos* is similar with them at first glance, as both output refined results progressively. However, these systems are optimized towards the most frequent queries, whereas the most valuable insights are more likely to be extracted from the tails of distributions. Furthermore, all these AQP systems execute each query in separate, which is not suitable for a multi-user fitting.

12 CONCLUSION

In this paper, we have presented *Davos*, Einblick's novel backend system. To overcome the shortcomings of existing data analytical systems, *Davos* combines multiple aspects, including progressive computation, approximate query processing and sampling. Moreover, considering the complexity of prescriptive analytics, *Davos* in particular focuses on supporting user-defined operations and has a latency-aware framework to optimize multi-tenant scenarios. Our evaluation shows that *Davos* greatly empowers users in making data driven decisions.

REFERENCES

[1] [n.d.]. Apache Arrow. https://arrow.apache.org/.

[2] [n.d.]. Apache Flink. https://flink.apache.org/.

[3] [n.d.]. Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[4] [n.d.]. Apache Storm. https://storm.apache.org/.

[5] [n.d.]. The BF Scheduler. https://en.wikipedia.org/wiki/Brain_Fuck_Scheduler.

[6] [n.d.]. Descriptive analytics 101: What happened? https://www.ibm.com/blogs/business-analytics/descriptive-analytics-101-what-happened/.

[7] [n.d.]. Einblick demo video. https://www.youtube.com/watch?v=4eb_idT4YrM.

[8] [n.d.]. Google Cloud. https://cloud.google.com/.

[9] [n.d.]. Monetdb. https://www.monetdb.org/.

[10] [n.d.]. Pandas. https://pandas.pydata.org/.

[11] Swarup Acharya, Phillip B Gibbons, and Viswanath Poosala. 2000. Congressional samples for approximate answering of group-by queries. In *Acm Sigmod Record*, Vol. 29. ACM, 487–498.

[12] Swarup Acharya, Phillip B Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. 1999. The Aqua approximate query answering system. In *ACM Sigmod Record*, Vol. 28. ACM, 574–576.

[13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 29–42.

[14] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[15] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *Nsdi*, Vol. 10. 20.

[16] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling udf-centric workflows. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1466–1477.

[17] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2015. Vizdom: interactive analytics through pen and touch. *Proceedings of the VLDB Endowment* 8, 12 (2015), 2024–2027.

[18] Philipp Eichmann, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2020. Idebench: A benchmark for interactive data exploration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1555–1569.

[19] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[20] Michael Hausenblas and Jacques Nadeau. 2013. Apache drill: interactive ad-hoc analysis at scale. *Big Data* 1, 2 (2013), 100–104.

[21] Joseph M Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J Haas. 1999. Interactive data analysis: The control project. *Computer* 32, 8 (1999), 51–59.

[22] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Acm Sigmod Record*, Vol. 26. ACM, 171–182.

[23] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2008. Scalable approximate query processing with the DBO engine. *ACM Transactions on Database Systems (TODS)* 33, 4 (2008), 23.

[24] Jaemin Jo, Wonjae Kim, Seunghoon Yoo, Bohyoung Kim, and Jinwook Seo. 2017. SwiftTuna: Responsive and incremental visual exploration of large-scale multidimensional data. In *2017 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE, 131–140.

[25] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. 2014. Distributed and interactive cube exploration. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 472–483.

[26] Zhicheng Liu and Jeffrey Heer. 2014. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2122–2131.

[27] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[28] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. AQP++: connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1477–1492.

[29] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[30] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 157–168.

[31] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM, 423–438.

[32] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. 2015. G-ola: Generalized on-line aggregation for interactive analysis on big data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 913–918.