KAIZENFLOW: A FRAMEWORK FOR HIGH-PERFORMANCE MACHINE LEARNING STREAM COMPUTING

GIACINTO PAOLO SAGGESE* AND PAUL SMITH*

Abstract.

Contents

1. Introduction

KaizenFlow is a framework to build, test, and deploy high-performance streaming computing systems based on machine learning and artificial intelligence.

The goal of KaizenFlow is to increase the productivity of data scientists by empowering them to design and deploy systems with minimal or no intervention from data engineers and devops support.

Guiding desiderata in the design of KaizenFlow include:

(1) Support rapid and flexible prototyping with the standard Python/Jupyter/data science tools
(2) Make it easy to replay stream events in testing and debugging
(3) Avoid software rewrites in going from prototype to production
(4) Specify through config
(5) Scale gracefully

These design principles are embodied in the many design features of KaizenFlow, which include:

(1) **Computation as a direct acyclic graph**. KaizenFlow represents models as direct acyclic graphs (DAG), which is a natural form for dataflow and reactive models typically found in real-time systems. Procedural statements are also allowed inside nodes.
(2) **Time series processing**. All KaizenFlow components (such as data store, compute engine, deployment) handle time series processing in a native way. Each time series can be univariate or multivariate (e.g., panel data) represented in a data frame format.

Date: December 11, 2023.

* Authors listed alphabetically.

With contributions from Shayan Ghasemnezhad, Danil Iachmenev, Tamara Jordania, Sonaal Kant, Samarth KaPatel, Grigorii Pomazkin, Sameep Pote, Juraj Smeriga, Daniil Tikhomirov, Nina Trubacheva, and Vladimir Yakovenko,

(3) **Support for both batch and streaming modes**. The framework allows running a model both in batch and streaming mode, without any change in the model representation. The same compute graph can be executed feeding data in one shot or in chunks (as in historical/batch mode), or as data is generated (as in streaming mode). KaizenFlow guarantees that the model execution is the same independently on how data is fed, as long as the model is strictly causal. Testing frameworks are provided to compare batch/streaming results so that any causality issues may be detected early in the development process.

(4) **Precise handling of time**. All components automatically track the knowledge time of when the data is available at both their input and output. This allows one to easily catch future peeking bugs, where a system is non-causal and uses data available in the future.

(5) **Observability and debuggability**. Because of the ability to capture and replay the execution of any subset of nodes, it is possible to easily observe and debug the behavior of a complex system.

(6) **Incremental computation and caching**. Because the dependencies between nodes are explicitly tracked by KaizenFlow, only nodes that see a change of inputs or in the implementation code need to be recomputed, while the redundant computation can be automatically cached.

(7) **Maximum parallelism**. Because the computation is expressed as a DAG, the KaizenFlow execution scheduler can extract the maximum amount of parallelism and execute multiple nodes in parallel in a distributed fashion, minimizing latency and maximizing throughput of a computation.

(8) **Configured through a hierarchical configuration**. Each parameter in a KaizenFlow system is controlled by a corresponding value in a configuration. In other words, the config space is homeomorphic with the space of KaizenFlow systems: each config corresponds to a unique KaizenFlow system, and vice versa, each KaizenFlow sytem is completely represented by a Config. A configuration is represented as a nested dictionary following the same structure of the DAG to make it easy to navigate its structure This makes it easy to create an ensemble of DAGs sweeping a multitude of parameters to explore the design space.

(9) **Tiling**. KaizenFlow's framework allows streaming data with different tiling styles (e.g., across times, across features, and both), to minimize the amount of working memory needed for a given computation, increasing the chances of caching computation.

(10) **Support for train/prediction mode**. A DAG can be run in 'fit' mode to learn some parameters, which are stored by the relevant DAG nodes, and then run in 'predict' mode to use the learned parameters to make predictions. This mimics the Sklearn semantic. There is no limitation to the number of evaluation phases that can be created (e.g., train, validation, prediction, save state, load state). Many different learning styles are supported from different types of runners (e.g., in-sample-only, in-sample vs out-of-sample, rolling learning, cross-validation).

(11) **Model serialization**. A fit DAG can be serialized to disk and then materialized for prediction in production.

(12) **Automatic vectorization**. KaizenFlow DAG nodes can apply a computation to a cross-section of features relying on numpy and Pandas vectorization.

(13) **Deployment and monitoring**. A KaizenFlow system can be deployed as a Docker container. Even the development system is run as a Docker container, supporting the development and testing of systems on both cloud (e.g., AWS) and local desktop. Airflow is used to schedule and monitor long-running KaizenFlow systems.

(14) **Python and Jupyter friendly**. The framework is completely implemented in high-performance Python. It supports natively 'asyncio' to overlap computation and I/O. The

same DAG can be run in a Jupyter notebook for research and experimentation or in a production script, without any change in code.

(15) **Python data science stack support**. Data science libraries (such as Pandas, numpy, scipy, sklearn) are supported natively to describe computation. The framework comes with a library of pre-built nodes for many ML/AI applications.

1.1. **Definition of stream computing.** In the computer science literature, several terms (such as event/data stream processing, graph computing, dataflow computing, reactive computing) are used to describe what in this paper we refer succintly to as "stream computing".

By stream processing we refer to a programming paradigm where streams of data (e.g., time series or dataframes) are the objects of computation. A series of operations (aka kernel functions) are applied to each element in the stream.

Stream computing represents a paradigm shift from traditional batch processing and imperative languages, emphasizing real-time data handling, adaptability, and parallel processing, making it highly effective for modern data-intensive applications.

The core principles of stream computing are:

(1) **Node-based architecture**. In stream and dataflow programming, the code is structured as a network of nodes. Each node represents a computational operation or a data processing function. Nodes are connected by edges that represent data streams.
(2) **Data-driven and reactive execution**. Execution in dataflow languages is data-driven, meaning that a node will process data as soon as it becomes available. Unlike imperative languages where the order of operations is predefined, in dataflow languages, the flow of data determines the order of execution.
(3) **Automatic parallelism**. The Dataflow programming paradigm naturally lends itself to parallel execution. Since nodes operate independently, they can process different data elements simultaneously, exploiting concurrent processing capabilities of modern hardware.
(4) **Continuous data streams**. Streams represent a continuous flow of data rather than discrete batches. Nodes in the network continuously receive, process, and output data, making them ideal for real-time data processing.
(5) **State management**. Nodes can be stateful or stateless. Stateful nodes retain information about previously processed data, enabling complex operations like windowing, aggregation, or pattern detection over time.
(6) **Compute intensity**. Stream processing is characterized by a high number of arithmetic operations per I/O and memory reference (e.g., it can be 50:1), since the same kernel is applied to all records and a number of records can be processed simultaneously. Furthermore, data is produced once and read only a few times.
(7) **Dynamic adaptability**. The dataflow model can dynamically adapt to changes in the data stream (like fluctuations in volume or velocity), ensuring efficient processing under varying conditions.
(8) **Scalability**. The model scales well horizontally, meaning one can add more nodes (or resources) to handle increased data loads without major architectural changes.
(9) **Event-driven processing**. Many dataflow languages support event-driven models where specific events in the data stream can trigger particular computational pathways or nodes.

1.2. **Applications of stream computing.** Stream computing is a natural solution in a wide range of industries and scenarios, as described below.

(1) Generic machine learning:
    - Graph-based computations are ubiquitous in modern deep learning.
(2) Financial Services

    - Trading: Analyzing market data in real-time to make automated trading decisions.
    - Fraud Detection: Monitoring transactions as they happen to detect and prevent fraudulent activities.
    - Risk Management: Real-time assessment of financial risks based on current market conditions and ongoing transactions.
3. Internet of Things (IoT):
    - Smart Homes: Processing data from various home devices for automation and monitoring.
    - Industrial IoT: Real-time monitoring and control of industrial equipment and processes.
    - Smart Cities: Integrating data from traffic, public services, and environmental sensors to optimize urban management.
4. Telecommunications:
    - Network Monitoring and Optimization: Analyzing traffic patterns to optimize network performance and detect anomalies.
    - Customer Experience Management: Real-time analysis of customer data to improve service and personalize offerings.
5. Healthcare:
    - Remote Patient Monitoring: Continuous monitoring of patient vitals for timely medical intervention.
    - Real-Time Health Data Analysis: Analyzing data streams from medical devices for immediate clinical insights.
6. Retail and E-Commerce:
    - Personalized Recommendations: Real-time analysis of customer behavior to offer personalized product recommendations.
    - Supply Chain Optimization: Streamlining logistics and inventory management based on real-time data.
7. Media and Entertainment:
    - Content Optimization: Real-time analysis of viewer preferences and behavior for content recommendations.
    - Live Event Analytics: Monitoring and analyzing data from live events for audience engagement and operational efficiency.
8. Transportation and Logistics:
    - Fleet Management: Tracking and managing vehicles in real time for optimal routing and scheduling.
    - Predictive Maintenance: Analyzing data from transportation systems to predict and prevent equipment failures.
9. Energy and Utilities:
    - Smart Grid Management: Balancing supply and demand in real-time and identifying grid anomalies.
    - Renewable Energy Optimization: Optimizing the output of renewable energy sources by analyzing environmental data streams.
10. Cybersecurity:
    - Intrusion Detection Systems: Real-time monitoring of network traffic to detect and respond to cyber threats.
    - Threat Intelligence: Analyzing global cyber threat data streams for proactive security measures.
11. Environmental Monitoring:
    - Climate and Weather Analysis: Processing data from environmental sensors for weather prediction and climate research.

    - Pollution Monitoring: Real-time tracking of air and water quality.

(12) Gaming:
    - In-Game Analytics: Real-time analysis of player behavior for game optimization and personalized experiences.

(13) Social Media Analytics:
    - Trend Analysis: Monitoring social media streams to identify and analyze trending topics and sentiments.

(14) Emergency Response:
    - Disaster Monitoring and Management: Real-time data analysis for effective response during natural or man-made disasters.

1.3. **Time series databases.** Data stores supporting stream computing are often specialized databases optimized for storing and managing time series data—data points indexed in time order. Time series data typically consists of sequences of values or events collected at regular or irregular intervals over time. These databases are designed to handle the unique challenges posed by time series data, including large volumes of data and real-time processing.

Some key aspects of time series databases:

1.3.1. *Time-indexed data time as a primary key.* In time series databases, time is often a primary axis, meaning that data is organized based on timestamps. The data usually consists of sequences of measurements taken over time, such as sensor data, stock market data, or server metrics.

1.3.2. *High-performance write and read efficient data ingestion.* Time series are optimized for fast writes, as time series data often involves high-frequency data collection (e.g., IoT sensors, financial tick data).

1.3.3. *Optimized query performance.* Time series databases are also optimized for time-based queries, such as aggregations over time intervals or retrieving data points within a specific time range.

2. Configuration layer

2.0.1. *Desirable properties of a configuration layer.* Configuring complex systems is not a trivial task. KaizenFlow uses the follow design principles for its configuration mechanisms. Some desirable properties for a configuration layer configuration to have are to be

- human-readable and as self-explanatory as possible
- version controllable, so that it can be tracked and reproduced
- corresponds to a unique system
- able to describe every system
- modular: a system composed of other systems should be described by the composition of the corresponding configurations
- executable: a configuration can be executed to generate the corresponding system
- declarative: the user describes how the system should look and other systems can build it

2.0.2. *KaizenFlow configuration layer.* All components in KaizenFlow are configured through a hierarchical data structure, called a `Config`.

A `Config`, together with a builder function, completely characterizes a KaizenFlow system. In other words, the builders represent the connectivity between components (which components need to be instantiated and how they need to be connected), while the `Config` represents the actual parameters needed to configure the components.

The builder function recursively creates and connects components using other builders, up to leaf objects, which are directly instantiated. The `Config` sets parameters in each object that can be accessed and controlled by the user.

The structure of a concrete Config follows the same organization of the components inside the KaizenFlow system. In our implementation parameters for each

2.0.3. *Example of KaizenFlow configuration layer.* Consider a system called `baz` that contains a component belonging to class `a`, which in turn includes a component `b`.

```python
class B:

  def __init__(self, param1: List[str], param2: int):
    ...


class FoobarA(AbstractA):

  def __init__(self, val1: int, val2: str, b: B):
    ...


def build_baz_v1(config: Config) -> AbstractA:
  """
  Build a System composed of a class derived from `AbstractA` and a class `B`.
  """
  # Build 'b'.
  b_obj = B(**config["a"]["b_ctor"])
  # Build 'a'.
  if config["a"]["type"] == "a_foobar":
    a_obj = FoobarA(**config["a"]["a_ctor"], b_obj)
  else:
    raise ValueError("Invalid a.type='%s'" % config["a"]["type"])
  return a_obj


config: Config = {
  "a": {
    {
      "type": "a_foobar",
      "a_ctor": {
        "val1": 42,
        "val2": "hello",
        "b_ctor": {
          "param1": ["foo", "bar"],
          "param2": -1,
        }
      }
    }
  }
```

Listing 1. Python example

This will build a system like:

3. DataPull

3.1. **DataPull principles.** - Adapters to different data sources

3.2. **DataPull data format.**

3.3. **Data/timing semantic.**

3.4. **KaizenFlow time series databases.** - knowledge time - different views of the data spliced together (e.g., historical and real-time)

4. DataFlow

4.1. **Computation as graphs.**

4.1.1. *DataFlow framework.* DataFlow is a computing framework to build and test AI and machine learning models that can run:

- with no changes in batch vs streaming mode
- in different modes, which trade off timing accuracy and speed (timed, non-timed, replayed simulation, and real-time execution)

The working principle underlying DataFlow is to run a model in terms of time slices of data so that both batch/historical and streaming/real-time semantics can be accommodated without any change in the model description.

Some of the advantages of the DataFlow approach are:

- Adapt a procedural description of a model to a reactive/streaming semantic
- Tiling to fit in memory
- Cached computation
- A clear timing semantic which includes support for knowledge time and detection of future peeking
- Ability to replay and debug model executions

A DAG Node has:

- inputs
- outputs
- a unique node id (aka `nid`)
- a (optional) state

Inputs and outputs to a DAG Node are dataframes, represented in the current implementation as `Pandas` dataframes. DAG node uses the inputs to compute the output (e.g., using `Pandas` and `Sklearn` libraries). A DAG node can execute in multiple "phases", referred to through the corresponding methods called on the DAG (e.g., `fit`, `predict`, `save_state`, `load_state`).

A DAG node stores an output value for each output and method name.

TODO(gp): circle with inputs and outputs

4.1.2. *DAG node examples.* Examples of operations that may be performed by nodes include:

- Loading data (e.g., market or alternative data)
- Resampling data bars (e.g., OHLCV data, tick data in finance)
- Computing rolling average (e.g., TWAP/VWAP, volatility of returns)
- Adjusting returns by volatility
- Applying EMAs (or other filters) to signals
- Performing per-feature operations, each requiring multiple features
- Performing cross-sectional operations (e.g., factor residualization, Gaussian ranking)
- Learning/applying a machine learning model (e.g., using sklearn)
- Applying custom (user-written) functions to data

Further examples include nodes that maintain relevant trading state, or that interact with an external environment:

- Updating and processing current positions
- Performing portfolio optimization

- Generating trading orders
- Submitting orders to an API

4.1.3. *DataFlow model.* A DataFlow model (aka `DAG`) is a direct acyclic graph composed of DataFlow nodes

It allows to connect, query the structure, ...

Running a method on a DAG means running that method on all its nodes in topological order, propagating values through the DAG nodes.

TODO(gp): Add picture.

4.1.4. *DagConfig.* A `Dag` can be built by assembling Nodes using a function representing the connectivity of the nodes and parameters contained in a `Config` (e.g., through a call to a builder `DagBuilder.get_dag(config)`).

A DagConfig is hierarchical and contains one subconfig per DAG node. It should only include `Dag` node configuration parameters, and not information about `Dag` connectivity, which is specified in the `Dag` builder part.

4.2. **Dataframe as unit of computation.** The basic unit of computation of each node is a "dataframe". Each node takes multiple dataframes through its inputs, and emits one or more dataframes as outputs.

In mathematical terms, a dataframe can be described as a two-dimensional labeled data structure, similar to a matrix but with more flexible features.

A Dataframe **df** can be represented as:

$$\mathbf{df} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

where:

- m is the number of rows (observations).
- n is the number of columns (variables).
- a_{ij} represents the element of the Dataframe in the i-th row and j-th column.

Some characteristics of dataframes are:

(1) Labeled Axes:
   - Rows and columns are labeled, typically with strings, but labels can be of any hashable type.
   - Rows are often referred to as indices and columns as column headers.

(2) Heterogeneous Data Types:
   - Each column j can have a distinct data type, denoted as $dtype_j$
   - Common data types include integers, floats, strings, and datetime objects.

(3) Mutable Size:
   - Rows and columns can be added or removed, meaning the size of $mathbfdf$ is mutable.
   - This adds to the flexibility compared to a traditional matrix.

(4) Alignment and Operations:
   - Dataframes support alignment and arithmetic operations along rows and columns.
   - Operations are often element-wise but can be customized with aggregation functions.

(5) Missing Data Handling:
   - Dataframes can contain missing data, denoted as `NaN` or a similar placeholder.
   - They provide tools to handle, fill, or remove missing data.

4.3. **DAG execution.**

4.3.1. *Simulation kernel.* A computation graph is a directed graph where nodes represent operations or variables, and edges represent dependencies between these operations.

For example, in a computation graph for a mathematical expression, nodes would represent operations like addition or multiplication, while edges would indicate which operations need to be completed before others.

The KaizenFlow simulation kernel schedules nodes according to their dependencies.

4.4. **Details about simulation kernel.** The most general case of simulation consists of multiple nested loops:

(1) **Multiple DAG computation**. The general workload contains multiple DAG computations, each one inferred through a `Config` belonging to a list of `Configs` describing the entire workload to execute.
   - In this set-up each DAG computation is independent, although some pieces of computations can be common across the workload. KaizenFlow will compute and then cache the common computations automatically as part of the framework execution

(2) **Learning pattern**. For each DAG computation, multiple train/predict loops represent different machine learning patterns (e.g., in-sample vs out-of-sample, cross-validation, rolling window)
   - This loop accommodates the need for nodes with state to be driven to learn parameters and hyperparameters and then use the learned state to predict on unseen data (i.e., out-of-sample)

(3) **Temporal tiling**. Each DAG computation runs over a tile representing an interval of time
   - As explained in section XYZ, KaizenFlow partition the time dimension in multiple tiles
   - Temporal tiles might overlap to accommodate the amount of memory needed by each node (see XYZ), thus each timestamp will be covered by at least one tile. In the case of DAG nodes with no memory, then time is partitioned in non-overlapping tiles.
   - The tiling pattern over time does not affect the result as long as the system is properly designed (see XYZ)

(4) **Spatial tiling**. Each temporal slice can be computed in terms of multiple sections across the horizontal dimension of the dataframe inputs, as explained in section XYZ.
   - This is constrained by nodes that compute features cross-sectionally, which require the entire space slice to be computed at once

(5) **Single DAG computation**. Finally a topological sorting based on the specific DAG connectivity is performed in order to execute nodes in the proper order. Each node executes over temporal and spatial tiles.

TODO(gp): Add picture showing the various loops

Note that it is possible to represent all the computations from the above loops in a single "scheduling graph" and use this graph to schedule executions in a global fashion.

Parallelization across CPUs comes naturally from the previous approach, since computations that are independent in the scheduling graph can be executed in parallel, as described in Section XYZ.

Incremental and cached computation is built-in in the scheduling algorithm since it's possible to memoize the output by checking for a hash of all the inputs and of the code in each node, as described in Section XYZ.

Even though each single DAG computation is required to have no loops, a System (see XYZ) can have components introducing loops in the computation (e.g., a Portfolio component in a trading system, where a DAG computes forecasts which are acted upon based on the available funds). In this case, the simulation kernel needs to enforce dependencies in the time dimension.

4.5. **Nodes ordering for execution.** TODO(gp, Paul): Extend this to the multiple loop.

Topological sorting is a linear ordering of the vertices of a directed graph such that for every directed edge from vertex u to vertex v, u comes before v in the ordering. This sorting is only possible if the graph has no directed cycles, i.e., it must be a Directed Acyclic Graph (DAG).

```
def topological_sort(graph):
    visited = set()
    post_order = []

    def dfs(node):
        if node in visited:
            return
        visited.add(node)
        for neighbor in graph.get(node, []):
            dfs(neighbor)
        post_order.append(node)

    for node in graph:
        dfs(node)

    return post_order[::-1]  # Reverse the post-order to get the topological order
```

4.6. **Heuristics for splitting code in nodes.** There are degrees of freedom in splitting the work between various nodes of a graph E.g., the same DataFlow computation can be described with several nodes or with a single node containing all the code

The trade-off is often between several metrics:

- Observability
  - More nodes make it easier to:
    - observe and debug intermediate the result of complex computation
    - profile graph executions to understand performance bottlenecks
- latency/throughput
  - More nodes:
    - allows for better caching of computation
    - allows for smaller incremental computation when only one part of the inputs change
    - prevents optimizations performed across nodes
    - incurs in more simulation kernel overhead for scheduling
    - allows more parallelism between nodes being extracted and exploited
- memory consumption
  - More nodes:
    - allows to partition the computation in smaller chunks requiring less working memory

A possible heuristic is to start with smaller nodes, where each node has a clear function, and then merge nodes if this is shown to improve performance

4.7. **DataFlow data format.** As explained in XYZ, raw data from `DataPull` is stored in a "long format", where the data is conditioned on the asset (e.g., full_symbol), e.g.,

```
                              full_symbol         open    high    low     close ...
timestamp
2021-09-01 00:00:00+00:00   binance::ADA_USDT   2.768   2.770   2.762   2.762
2021-09-01 00:00:00+00:00   binance::AVAX_USDT  39.510  39.540  39.300 39.320
2021-09-01 00:00:00+00:00   binance::ADA_USDT   2.763   2.765   2.761   2.764
```

`DataFlow` represents data through multi-index dataframes, where:

- the index is a full timestamp
- the outermost column index is the "feature"
- the innermost column index is the asset, e.g.,

```
                                                             close            high
                             binance::ADA_USDT   binance::AVAX_USDT            ...
timestamp
2021-09-01 00:00:00+00:00                2.762                39.32
2021-09-01 00:00:00+00:00                2.764                39.19
```

The reason for this convention is that typically features are computed in a uni-variate fashion (e.g., asset by asset), and DataFlow can vectorize computation over the assets by expressing operations in terms of the features. E.g., we can express a feature as

```
df["close", "open"].max() - df["high"]).shift(2)
```

A user can work with DataFlow at 4 levels of abstraction:

(1) Pandas long-format (non multi-index) dataframes and for-loops
    - We can do a group-by or filter by full_symbol
    - Apply the transformation on each resulting dataframe
    - Merge the data back into a single dataframe with the long-format
(2) Pandas multiindex dataframes
    - The data is in the DataFlow native format
    - We can apply the transformation in a vectorized way
    - This approach is best for performance and with compatibility with DataFlow point of view
    - An alternative approach is to express multi-index transformations in terms of approach 1 (i.e., single asset transformations and then concatenation). This approach is functionally equivalent to a multi-index transformation, but typically slow and memory inefficient
(3) DataFlow nodes
    - A DataFlow node implements certain transformations on dataframes according to the DataFlow convention and interfaces
    - Nodes operate on the multi-index representation by typically calling functions from level 2 above
(4) DAG
    - A series of transformations in terms of DataFlow nodes

An example ./amp/dataflow/notebooks/gallery_dataflow_example.ipynb TODO(gp): Fix this reference.

4.8. **KaizenFlow System.**

4.8.1. *Motivation.* While KaizenFlow requires a DAG should not have cycles, general computing systems might need to reuse the state from computation performed on past data. E.g., in a trading system, there is often a Forecast component that can be modeled as a DAG with no cycles and a

Portfolio object that uses the forecasts to compute the desired allocation of capital across different positions based on the previous positions.

KaizenFlow supports this need by assembling multiple DAGs into a complete `System` that allows cycles.

The assumption is that DAGs are computationally expensive, while other components mainly execute light procedural computation that requires interaction with external objects such as databases, filesystems, sockets.

TODO(gp): Add picture

TODO(gp): Explain that System are derived from other Python objects.

4.9. **Timing semantic and clocks.**

4.10. **Batch and streaming mode using tiling.**

4.10.1. *The property of tilability.*

The working principle of a DataFlow computation is that nodes should be able to compute their outputs from their inputs without a dependency on how the inputs are partitioned along the dataframe axes of the inputs (e.g., the time and the feature axes). When this property is valid we call a computation "tilable".

A slightly more formal definition is that a computation $f()$ is tilable if:

$$f(dfX \cup dfY) = f(dfX) \cup f(dfY)$$

where:

- dfX and dfY represent input data frames (which can optionally) overlap
- $\cup$ is an operation of concat along consecutive
- $f()$ is a node operation

TODO(gp): In reality the property requires that feeding data, computing, and then filtering is invariant, like

f(A, B)

$$\forall t1 \leq t2, t3 \leq t4 : \exists T : f(A[t1{-}T : t2] \cup A[t3{-}T : t4])[t1 : t4] = f(A[t1{-}T : t2])[t1 : t4] \cup f(A[t3{-}T : t4])[t1 : t4]$$

This property resembles linearity in the sense that a transformation $f()$ is invariant over the partitioning of the data.

A sufficient condition for a DAG computation to be tileable, is for all the DAG nodes to be tileable. The opposite is not necessarily true, and no interest in general, since we are interested in finding ways to describe the computation so that it is tileable.

A node that has no memory, e.g., whose computation

Nodes can have "memory", where the output for a given tile depends on previous tile. E.g., a rolling average has memory since samples with different timestamps are combined together to obtain the results. A node with finite memory is always tileable, while nodes with infinite memory are not necessarily tileable. If the computation can be expressed in a recursive form across axes (e.g., an exponentially weighted moving average for adjacent intervals of times), then it can be made tileable by adding auxiliary state to store the partial amount of computatin.

4.10.2. *Temporal tiling.*

In most computations there is a special axis that represents time and moves only from past to future. The data along other axes represent (potentially independent) features.

This is an easy requirement to enforce if the computation has no memory, e.g., in the following example of Python code using Pandas

```
df1 =
                          a         b
2023-01-01 08:00:00   10        20
2023-01-01 08:30:00   10        20
2023-01-01 09:00:00   10        20
2023-01-01 09:30:00   10        20

df2 =
                          a         b
2023-01-01 08:00:00   10        20
2023-01-01 08:30:00   10        20
2023-01-01 09:00:00   10        20
2023-01-01 09:30:00   10        20

dfo = df1 + df2
```

DataFlow can partition df1 and df2 in different slices obtaining the same result, e.g.,

```
df1_0 =
                          a         b
2023-01-01 08:00:00   10        20
2023-01-01 08:30:00   10        20

df2_0 =
                          a         b
2023-01-01 08:00:00   10        20
2023-01-01 08:30:00   10        20

dfo_0 = df1_0 + df2_0


df1_1 =
                          a         b
2023-01-01 09:00:00   10        20
2023-01-01 09:30:00   10        20

df2_1 =
                          a         b
2023-01-01 09:00:00   10        20
2023-01-01 09:30:00   10        20

dfo_1 = df1_1 + df2_1

dfo = pd.concat([dfo_1, dfo_2])
```

Consider the case of a computation that relies on past values

```
dfo = df1.diff()
```

This computation to be invariant to slicing needs to be fed with a certain amount of previous data

```
df1_0 =
                          a         b
```

```
2023-01-01 08:00:00   10        20
2023-01-01 08:30:00   10        20

dfo_0 = df1_0.diff()
dfo_0 = dfo_0["2023-01-01 08:00:00":"2023-01-01 08:30:00"]

df1_1 =
                      a         b
2023-01-01 08:30:00   10        20
2023-01-01 09:00:00   10        20
2023-01-01 09:30:00   10        20

dfo_1 = df1_1.diff()
dfo_1 = dfo_1["2023-01-01 09:00:00":"2023-01-01 09:30:00"]

dfo = pd.concat([dfo_1, dfo_2])
```

In general as long as the computation doesn't have infinite memory (e.g., an exponentially weighted moving average)

This is possible by giving each nodes data that has enough history

Many interesting computations with infinite memory (e.g., EMA) can also be decomposed in tiles with using some algebraic manipulations - TODO(gp): Add an example of EMA expressed in terms of previous

Given the finite nature of real-world computing (e.g., in terms of finite approximation of real numbers, and bounded computation) any infinite memory computation is approximated to a finite memory one. Thus the tiling approach described above is general, within any desired level of approximation.

The amount of history is function of a node

4.10.3. *Cross-sectional tiling.*

- The same principle can be applied to tiling computation cross-sectionally
- Computation that needs part of a cross-section need to be tiled properly to be correct
- TODO(gp): Make an example

4.10.4. *Temporal and cross-sectional tiling.*

- These two styles of tiling can be composed
- The tiling doesn't even have to be regular, as long as the constraints for a correct computation are correct

4.10.5. *Detecting incorrect tiled computations.*

- One can use the tiling invariance of a computation to verify that it is correct
- E.g., if computing a DAG gives different results for different tiled, then the amount of history to each node is not correct

4.10.6. *Benefits of tiled computation.*

- Another benefit of tiled computation is that future peeking (i.e., a fault in a computation that requires data not yet available at the computation time) can be detected by streaming the data with the same timing as the real-time data would do

A benefit of the tiling is that the compute frame can apply any tile safe transformation without altering the computation, e.g.,

- vectorization across data frames

- coalescing of compute nodes
- coalescing or splitting of tiles
- parallelization of tiles and nodes across different CPUs
- select the size of a tile so that the computation fits in memory

4.10.7. *Batch vs streaming.* Once a computation can be tiled, the same computation can be performed in batch mode (e.g., the entire data set is processed at once) or in streaming mode (e.g., the data is presented to the DAG as it becomes available) yielding the same result

This allows a system to be designed only once and be run in batch (fast) and real-time (accurate timing but slow) mode without any change

In general the more data is fed to the system at once, the more likely is to being able to increase performance through parallelization and vectorization, and reducing the overhead of the simulation kernel (e.g., assembling/splitting data tiles, bookkeeping), at the cost of a larger footprint for the working memory

In general the smaller the chunks of data are fed to the system (with the extreme condition of feeding data with the same timing as in a real-time set-up), the more unlikely is a fault in the design it is (e.g., future peeking, incorrect history amount for a node)

Between these two extremes is also possible to chunk the data at different resolutions, e.g., feeding one day worth of data at the time, striking different balances between speed, memory consumption, and guarantee of correctness

4.11. **Vectorization.**

4.11.1. *Vectorization.* Vectorization is a technique for enhancing the performance of computations by simultaneously processing multiple data elements with a single instruction, leveraging the capabilities of modern processors (e.g., SIMD (Single Instruction, Multiple Data) units).

4.11.2. *Vectorization in KaizenFlow.* Given the DataFlow format, where features are organized in a hierarchical structure, KaizenFlow allows to apply an operation to be applied across the cross-section of a dataframe. In this way KaizenFlow exploits Pandas and NumPy data manipulation and numerical computing capabilities, which are in turns built on top of low-level libraries written in languages like C and Fortran. These languages provide efficient implementations of vectorized operations, thus bypassing the slower execution speed of Python loops.

4.11.3. *Example of vectorized node in KaizenFlow.* TODO

4.12. **Incremental, cached, and parallel execution.**

4.12.1. *DataFlow and functional programming.* The DataFlow computation model shares many similarity with functional programming:

- Data immutability: data in dataframe columns is typically added or replaced. A node in a DataFlow graph cannot alter data in the nodes earlier in the graph.
- Pure functions: the output of a node depends only on its input values and it does not cause observable side effects, such as modifying a global state or changing the value of its inputs
- Lack of global state: nodes do not rely on data outside their scope, especially global state

4.12.2. *Incremental computation.* Only parts of a compute graph that see a change of inputs need to be recomputed.

Incremental computation is an approach where the result of a computation is updated in response to changes in its inputs, rather than recalculating everything from scratch

4.12.3. *Caching.* Because of the ”functional” style (no side effects) of data flow, the output of a node is determinstic and function only of its inputs and code.

Thus the computation can be cached across runs. E.g., if many DAG simulations share the first part of simulation, then that part will be automatically cached and reused, without needing to be recomputed multiple times.

4.12.4. *Parallel execution.* Parallel and distributed execution in KaizenFlow is supported at two different levels:

- Across runs: given a list of `Config`, each describing a different system, each simulation can be in parallel because they are completely independent.
- Intra runs: each DataFlow graph can be run exploiting the fact that nodes

In the current implementation for intra-run parallelism Kaizen flow relies on `Dask` For across-run parallelism KaizenFlow relies on `joblib` or `Dask`

Dask extends the capabilities of the Python ecosystem by providing an efficient way to perform parallel and distributed computing.

Dask supports various forms of parallelism, including multi-threading, multi-processing, and distributed computing. This allows it to leverage multiple cores and machines for computation.

When working in a distributed environment, Dask distributes data and computation across multiple nodes in a cluster, managing communication and synchronization between nodes. It also provides resilience by re-computing lost data if a node fails.

4.13. **Train and predict.**

4.13.1. *Stateful nodes.* A DAG node is stateful if it uses data to learn parameters (e.g., linear regression coefficients, weights in a neural network, support vectors in a SVM) during the `fit` stage, that are then used in a successive `predict` stage.

The state is stored inside the implementation of the node.

The state of stateful DAG node varies during a single simulation.

TODO: Add snippet of code showing stateful node.

4.13.2. *Stateless nodes.* A DAG node is stateless if the output is not dependent on previous `fit` stages. In other words the output of the node is only function of the current inputs and of the node code, but not from inputs from previous tiles of inputs.

A stateless DAG node emits the same output independently from the current and previous `fit` vs `predict` phases.

A stateless DAG node has no state that needs to be stored across a simulation.

4.13.3. *Loading and saving node state.* Each stateful node needs to allow saving and loading its state on demand of the framework.

A stateless node should emit an empty state when saving and assert in case a non-empty state is presented during a load phase.

KaizenFlow simulation kernel orchestrate loading and saving nodes for an entire DAG to serialize and deserialize a DAG to disk.

TODO(gp): Add an example of data layout

KaizenFlow allows to load the state of a DAG for further analysis. E.g., one might want to see how weights of a linear model evolve over time in a rolling window simulation.

4.13.4. *Different types of learning.* Cross-validation is a statistical method used to estimate the skill of machine learning models. It is primarily used to assess how the results of a statistical analysis will generalize to an independent data set. In the context of time series data, where the temporal order of data points is important, special forms of cross-validation, like in-sample and rolling-window cross-validation, are used. Here's a brief explanation of these methods:

In-Sample Cross-Validation:

In this method, the entire dataset is used for both training and testing. The model is trained on a certain portion of the data (the training set) and then tested on the remaining data (the test set). One common approach is to split the data chronologically. For example, the model might be trained on the first 80In-sample cross-validation is useful for time series data because it respects the chronological order of observations. Rolling-Window (or Walk-Forward) Cross-Validation:

This method is more sophisticated and particularly suited for time series data. In rolling-window cross-validation, the model is trained on a fixed-size window of data and then makes predictions for the subsequent time period. After each training and testing phase, the window is "rolled" forward, which means that the model is retrained on a new window of data including the most recent observations. For example, if you have monthly data for 10 years, you might train the model on the first year of data and test it on the next month. Then, you roll the window forward by one month (so the training data now starts from month 2 and goes up to month 13) and test on the 14th month. This process continues until you have tested the model on all available data. This method is especially useful for evaluating the model's performance over time and for datasets where the relationship between input and output variables changes. Both methods have their advantages and are chosen based on the specific characteristics of the dataset and the research or business problem. In-sample cross-validation is simpler and can be a good choice when the dataset is not very large, while rolling-window cross-validation is more robust for evaluating time-series models as it mimics the real-world scenario of training a model on past data and predicting future events.

4.14. **Observability and debuggability.**

4.14.1. *Running a DAG partially.* KaizenFlow allows to run nodes and DAGs in a notebook during design, analysis, and debugging phases, and in a Python script during simulation and production phases.

It is possible to run a DAG up to a certain node to iterate on its design and debug.

TODO: Add example

4.14.2. *Replaying a DAG.* Each DAG node can:

- capture the stream of data presented to it during either a simulation and real-time execution
- serialize the inputs and the outputs, together with the knowledge timestamps
- play back the outputs

KaizenFlow allows to describe a cut in a DAG and capture the inputs and outputs at that interface. In this way it is possible to debug a DAG replacing all the components before a given cut with a synthetic one replaying the observed behavior together with the exact timing in terms of knowledge timestamps.

This allows to easily:

- capture failures in production and replay them in simulation for debugging
- write unit tests using observed data traces

KaizenFlow allows each node to automatically save all the inputs and outputs to disk to allow replay and analysis of the behavior with high fidelity.

4.15. **Profiling DataFlow execution.**

4.16. **DataFlow and the Python data stack.**

5. ML Ops

6. Application of KaizenFlow to quant finance

7. Comparison to other computing framework

7.1. **Comparison principles.**

7.1.1. *Static vs Dynamic.* Static Nature: TensorFlow uses a static computational graph, meaning the graph is defined before it is run.

Dynamic Nature: Unlike some other frameworks that use static computational graphs, PyTorch operates on a dynamic (or "eager") computational graph. This means the graph is built on-the-fly as operations are executed. This property is known as the "define-by-run" paradigm.

7.1.2. *Fundamental data structure.* Tensors: The fundamental data structure in PyTorch is the Tensor, which is similar to NumPy arrays but with additional capabilities to operate on GPUs for accelerated computing.

7.1.3. *Python support.* Integration with NumPy, Pandas

Users can extend its functionalities using Python's features, and it allows for the integration of other Python libraries.

7.1.4. *Native time series support.* Knowledge time

7.1.5. *Support for general type of computation.*

7.1.6. *Parallel Processing.* Enables parallel processing of large datasets by breaking them into smaller, manageable chunks. It executes operations on these chunks in parallel, utilizing multiple cores on a single machine or across a cluster of machines.

7.1.7. *Distributed Computation.* Distribute computations over a cluster, making it suitable for both local processing on a single machine and large-scale computations on distributed resources.

7.1.8. *Support for streaming.*

7.2. **Pytorch.** PyTorch is an open-source machine learning library. It is widely used for deep learning applications and is known for its ease of use, flexibility, and dynamic computational graph.

Fundamental data structure: tensors

7.3. **TensorFlow.** Fundamental data structure: tensors

7.4. **Dask.** Fundamental data structure: dataframe, array, sets

7.5. **Apache Spark.**

7.6. **Apache Flink.**

7.7. **Google DataFlow.** - The focus of DataFlow is on events - KaizenFlow focuses on dense computation

References

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle, *The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, out-of-Order Data Processing* **8** (2015), no. 12, 1792-1803, DOI 10.14778/2824032.2824076.

[2] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas, *Apache Flink: Stream and Batch Processing in a Single Engine*, IEEE Data Engineering Bulletin **38** (2015).

[3] Wes McKinney, *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 2010, pp. 56-61, DOI 10.25080/Majora-92bf1922-00a.

[4] Eman Shaikh, Iman Mohiuddin, Yasmeen Alufaisan, and Irum Nahvi, *Apache Spark: A Big Data Processing Engine*, posted on 2019, 1-6, DOI 10.1109/MENACOMM46666.2019.8988541.

[5] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica, *Spark: Cluster Computing with Working Sets*, posted on 2010, 10, DOI 10.5555/1863103.1863113.