GraphReduce: scalable feature engineering with graphs

Wes Madrigal
Kurve, Inc.
wes@kurve.ai

ABSTRACT

GraphReduce is a programming model and associated software abstractions for performing computational operations on many disparate tabular datasets and merging them together, usually for ML/AI feature engineering. Despite the many advances in Transformers and LLMs, performing applied machine learning on tabular datasets remains a challenge due to the heavy lifting involved in feature engineering. Oftentimes multiple features share the same grouping and should be computed in tandem in a single *GroupBy* operation, unlike how feature stores abstract the problem.
GraphReduce represents tables as nodes in a graph and relationships as edges. Users define node-level operations on individual tables and edge-level operations which are conditioned on two or more tables being merged, such as multiplying two column values from two different tables. By leveraging a graph-based approach to model tabular data we can elegantly handle data duplication that results from one to many relationships by performing a granularity directed depth-first traversal and recursive rollup of the data to a root dimension of choice (e.g., customer-level). This allows computational graphs that involve many tables to maintain a similar level of complexity through a shared and extensible interface. Ultimately, building features at different dimensions with varying target variables becomes as simple as changing a few top-level parameters. GraphReduce provides an interface for performing entirely automated feature engineering, similar to those proposed in prior works. We extend these works and add additional extensibility by making the computational layer a parameter on the top-level GraphReduce object, allowing for interoperability with various computational backends such as pandas, polars, spark, Snowflake and others. Additionally, in prior works there is an assumption that the target variable for a machine learning problem is connected to the dimension at which the problem is modeled. This is often not the case, as we show, and GraphReduce provides abstractions for computed target variables across data dimensions.

1. INTRODUCTION

Analytics and AI applications are increasing in demand, yet the data preparation and feature engineering workloads required are often implemented in a one off manner. As a result, technical debt in analytics and AI data preparation code accumulates. This type of technical debt is often referred to as “pipeline jungles” because it is typically composed of many sequential functions spread across files or modules. Data workloads for analytics and AI often involve joining multiple tables

together, performing pre/post join operations, and dealing with time. At the time of writing, no machine learning model can learn directly on data stored across multiple disparate tables. Machine learning models require data to be joined into a single table at a single granularity and prepared for modeling through a process of joins, filters, normalizations/imputation, annotations, aggregations/reductions and broadly what is referred to as “feature engineering”. Most complex analytics and AI data workloads share a similar sequence of high-level operations, which allow for abstractions.

GraphReduce represents tables as nodes of a graph, relationships between tables as edges in the graph, and top-level parameters at the graph-level. We leverage graph data structure abstractions from networkx and implement domain-specific abstractions for joining tables, filtering, annotating, normalizing, aggregating/reducing, and dealing with time. In addition, the computational layer is parameterizable, which allows for swapping between different computational backends (e.g., Spark, pandas, dask).

Most ML/AI models built on tabular data require the dataset be aggregated to a particular dimension (e.g., customer-level) so there isn’t duplicate data. For example, if a data scientist is building a churn prediction model for customers, all relational datasets being used must be aggregated, or reduced, to the granularity of the customer data. Additionally, if the data is time series all training and test data throughout each table involved must use the same dates. We will use the following tables as an example throughout the paper:

- Customers
- notifications
- Orders
- Order_products
- Notification_interactions
- notification_interaction_types

GraphReduce provides an abstraction for automated feature engineering, extending previous work such as the DSM [3] and One Button Machine [4]. The disadvantage of DSM is that it does not allow for custom implementations in the computational graph nor does it allow for pluggable computational layers (e.g., using Spark vs. pandas). One Button Machine allows for the utilization of Spark to deal with massive scale, addressing a shortcoming of DSM, but does not allow for custom implementation of data transformation logic nor utilization of external libraries. We overcame two shortcomings of prior work: lack of swappable computational layers and lack of customizability. In GraphReduce users can parameterize the computational layer and customize every type of operation at the node-level.

2. RELATED WORK

Feature engineering automation and abstraction can be primarily bucketed under *feature engineering* or *automatic feature engineering*.

A. Automatic feature engineering

Data Science Machine (DSM) [3] automated feature engineering with aggregation primitives for specific data types. They were some of the first to do so. There are three main disadvantages of DSM: the type-based aggregation primitives are limited, there is no way of swapping the computational layer, and there is no ability to customize a particular datasets’ operations. GraphReduce addresses the latter two of these issues, allows for the user to extend the type-based aggregation primitives, but doesn’t automate it.

One Button Machine (OneBM) [4] extended DSM and brought additional capabilities such as enhanced aggregation primitives to allow support of more diverse data and introduced distributed computing via Spark. OneBM also suffers from the lack of swappable computational layers and no ability to customize operations for a particular dataset in the computational graph.

Graph-based Feature Synthesis (GFS) [10] and other GNN-based approaches are so architecturally different that we didn't compare our work with theirs. That being said, this area also suffers from a lack of customizability for issues that creep up in automated data systems such as data duplication.

3. ABSTRACTIONS

We can represent tables as nodes of a graph and the relationships as edges in the graph. Join operations are handled over edges in the graph in a directionally aware way (e.g., depth-first). This allows us to deal with cardinality in an elegant way by traversing to the most granular nodes and rolling up to the root while reducing along the way.

3.1 GraphReduce object

A compute graph defined with graphreduce requires the instantiation of a GraphReduce
instance, definition and instantiation of one or more nodes, and definition of the edges in the graph. The graphreduce instance needs, at minimum, the following parameters:

- a root node, which is the dimension to which all data will be reduced/aggregated
- a compute layer, such as pandas or Spark

For more advanced compute graphs, such as those involving time and target variable generation, the following parameters will be needed:

- A cut date, which serves as the data to filter data around
- A compute period, which indicates how much history to include
- A label period, which indicates, if generating a forecast label, the time period for the forecast
- A label node, which indicates the relational table on which the label gets computed
- A label field, which indicates the relational table's field that is used to compute the label

The GraphReduce class is instantiated with the parameters above. When automated feature engineering is used the developer should specify the number of front and back hops to constrain the computation graph. The central method in the GraphReduce instance that executes the entire computational graph is called *do_transformations*. When calling this method the following sequence of operations will occur:

1. Check that all nodes have a unique prefix (e.g., cust_[X] for all customer columns).
2. Push down all top-level parameters through the graph, such as the computational layer, cut dates, compute periods, label periods.
3. Loop through the graph and run *do_filters, do_annotate, and do_normalize*

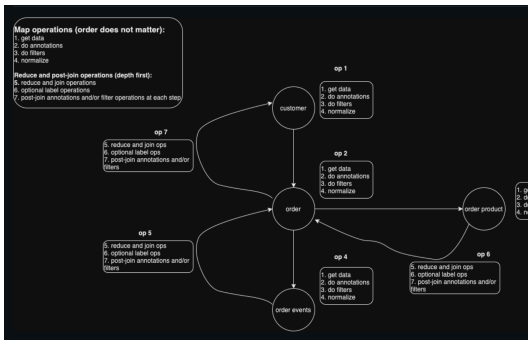
4. Loop through the graph in a depth-first way for each parent, child relationship.
5. Check if the child node should be reduced (boolean) and, if so, reduce.
6. Join the child node to the parent node.
7. 

Figure 1

3.2 Nodes

Nodes must subclass from the GraphReduceNode parent class and define the following abstract methods: *do_filters, do_annotate, do_normalize, do_reduce, do_post_join_annotate, do_labels*. GraphReduce takes a convention over configuration approach to maintain a consistent interface across computational graphs. A node must be instantiated with the following parameters:

- The path to the dataset being loaded.
- A prefix for the data originating from this dataset.
- A primary key, which is one of the columns in the dataset.
- If operations must consider time, a date key must be parameterized.

The standard Node class is designed for custom implementations and for automated feature engineering compute graphs the DynamicNode must be used.

3.3 Dynamic Nodes

The dynamic node allows for automated feature engineering. To use automated feature engineering the *auto_features* parameter must be set to True on the GraphReduce object. To use a dynamic node one needs only specify the primary key, date key, data path, and a prefix. If the relationships are known the user can loop through relationships and dynamically build up a compute graph with GraphReduce by using these dynamic nodes.

3.4 Edges

Edges in graphreduce are defined by specifying the parent node, parent key, relation node, relation key, and a boolean specifying whether to reduce the child node or join it as is. The parent node is the node to which the relation node will be joined. The parent key and relation key are the keys through which the join will happen. The reduce boolean must be specified if aggregation/reduce operations are defined on the relation node, and in cases where there is a one to many relationship between the nodes it is recommended to reduce data. The API for adding edges is similar to that of *networkx*.

Example (and figures)

Take an example dataset with 6 CSV files: cust.csv, orders.csv, order_products.csv, notifications.csv, notification_interactions.csv, and notification_interaction_types.csv. The relationships between these tables are known with 5 relationships. The plot_graph function in graphreduce renders the figure in Figure 2. To define a computational graph

for this dataset we need to instantiate a *GraphReduce* instance, instantiate each table as a node, and specify the 5 relationships as edges. After we define the computational graph we can execute it with *GraphReduce.do_transformations()*. To see the most up-to-date API and docs please visit the github README: https://github.com/wesmadrigal/graphreduce.

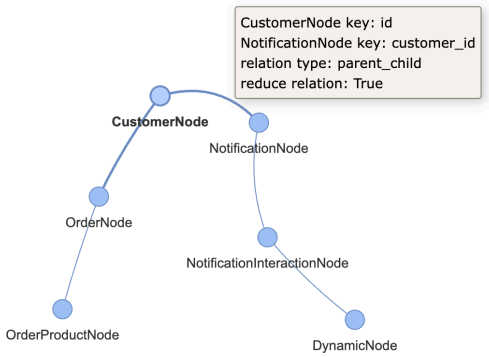


Figure 2

3.5 Compute and Storage layers

At the time of writing the supported computation layers are: pandas, dask, and Spark. There are near-term plans to expand to SQL dialects such as Snowflake and Redshift. Since graphreduce only concerns itself with computational graphs there is minimal effort around storage. However, a storage client is provided to offload checkpoints throughout the computation. For example, if there were a 5-node computational graph and the developer wanted to checkpoint the data at node 3 (e.g., *customers*) and method 2 (e.g., *do_annotate)*, that is made simple through an object and python wrapper methods.

3.6 Automated Feature Engineering

Graphs with only DynamicNodes are what is needed for automated feature engineering. In order to define one in graphreduce we need a data structure that contains the tables of interest and their relationships. With said data structure we can programmatically loop through and instantiate DynamicNodes. With a graphreduce instance and the root node, target node, target field, any time-related parameters specified we can then add the edges. Finally, if desired, the maximum number of forward and backward hops can be specified to constraint the computational graph from bringing distant data back to the root node. The ability to rapidly iterate through different shaped ML-ready feature vectors with different root nodes, targets, and time slices allows for rapid prototyping and deployment of ML models and complex analytics.

For any given database, data warehouse, or data lake the goal is conditioning a target variable *y* on all relevant data *x*. As different *y* targets are selected the tables involved in generating *x* are prone to change, meaning efficient abstractions for easily reshaping *x* are of high interest. Ideally we have *P(Y | Entire Database)*. Historically that has been computationally infeasible in terms of compute power and human capital, but maybe GraphReduce brings us one step closer.

4. CONCLUSIONS

The abstractions provided and included in GraphReduce allow for large, complex computational graphs to be more

easily defined and composed. These abstractions are being used in production settings at various companies today with proven success dealing with feature engineering complexity in MLOps pipelines. We extend prior works by allowing for customization of table/node-level data transformation definitions and make the computational layer being used a parameterizable aspect of the design, unlike prior works. The complexities of multiple data granularities, time travel, and others are all abstracted away but still controllable through top-level parameters in the interface.

REFERENCES

[1] Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation* (OSDI '04).

[2] Bernays, P., Schönfinkel, M. Zum Entscheidungsproblem der mathematischen Logik. *Math. Ann.* **99**, 342–372 (1928).

[3] Kanter, M., & Veeramachaneni, K. (2015). Deep feature synthesis: Towards automating data science endeavors. In *Proceedings of the IEEE International Conference on Data Science and Advanced Analytics* (DSAA, 2015).

[4] Lam, H. T., Thiebaut, J.-M., Sinn, M., Chen, B., Mai, T., & Alkan, O. (2017). One button machine for automating feature engineering in relational databases.

[5] Hagberg, Aric, Swart, Pieter J., and Schult, Daniel A. *Exploring network structure, dynamics, and function using NetworkX*. United States: N. p., 2008. Web.

[6] J. M. Kanter and K. Veeramachaneni (2015). Deep feature synthesis: Towards automating data science endeavors.

[7] U. Khurana, D. Turaga, H. Samulowitz, and S. Parthasarathy, editors. *Cognito: Automated Feature Engineering for Supervised Learning , ICDM 2016*.

[8] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2016). Apache Spark: A unified engine for big data processing.

[9] Biem, A., Butrico, M., Feblowitz, M., Klinger, T., Malitsky, Y., Ng, K., Perer, A., Reddy, C., Riabov, A., Samulowitz, H., Sow, D., Tesauro, G., & Turaga, D. (2015). Towards Cognitive Automation of Data Science. *Proceedings of the AAAI Conference on Artificial Intelligence*, *29*(1).

[10] Rau, J. (2023). Fermionization in Mean-Field Theory: Slater Determinants as Minimum-Uncertainty States.