Optimizing Data Pipelines for Machine Learning in Feature Stores

Rui Liu[1], Kwanghyun Park[2], Fotis Psallidas[3], Xiaoyong Zhu[3], Jinghui Mo[4], Rathijit Sen[3], Matteo Interlandi[3], Konstantinos Karanasos[5], Yuanyuan Tian[3], Jesús Camacho-Rodríguez[3]
[1]University of Chicago, [2]Yonsei University, [3]Microsoft, [4]LinkedIn, [5]Meta
[1]ruiliu@cs.uchicago.edu, [2]kwanghyun.park@yonsei.ac.kr, [3]{firstname.lastname}@microsoft.com, [4]jmo@linkedin.com, [5]kkaranasos@meta.com

ABSTRACT

Data pipelines (i.e., converting raw data to features) are critical for machine learning (ML) models, yet their development and management is time-consuming. Feature stores have recently emerged as a new "DBMS-for-ML" with the premise of enabling data scientists and engineers to define and manage their data pipelines. While current feature stores fulfill their promise from a functionality perspective, they are resource-hungry—with ample opportunities for implementing database-style optimizations to enhance their performance. In this paper, we propose a novel set of optimizations specifically targeted for point-in-time join, which is a critical operation in data pipelines. We implement these optimizations on top of Feathr: a widely-used feature store, and evaluate them on use cases from both the TPCx-AI benchmark and real-world online retail scenarios. Our thorough experimental analysis shows that our optimizations can accelerate data pipelines by up to 3× over state-of-the-art baselines.

PVLDB Reference Format:
Rui Liu, Kwanghyun Park, Fotis Psallidas, Xiaoyong Zhu, Jinghui Mo, Rathijit Sen, Matteo Interlandi, Konstantinos Karanasos, Yuanyuan Tian, Jesús Camacho-Rodríguez. Optimizing Data Pipelines for Machine Learning in Feature Stores. PVLDB, 16(13): 4230 - 4239, 2023.
doi:10.14778/3625054.3625060

1 INTRODUCTION

Nowadays, it is common that organizations deploy hundreds if not thousands of machine learning (ML) models to power important applications such as search, recommendation systems, or ads placement [27]. Data pipelines for ML, which usually involve data ingestion, feature engineering, and other preprocessing steps to convert raw source data into a format that can be used for training ML models, are critical in building these ML applications. Improperly implemented pipelines can result in correctness issues that decrease the accuracy of the ML models being deployed. One of the most common, non-obvious, and destructive issues is data leakage [47], which occurs when a form of the target variable, also known as a *label*, is involved in the feature set used for training but is unavailable during the inference stage. Unfortunately, as more applications are developed over time, and thus more features need to be extracted, the complexity of developing and managing the *ML pipeline jungles* [52] that compute these features increases rapidly, and with it the possibility of encountering these pitfalls.

In this context, feature stores (FSs) have been proposed to provide a common interface to compute, store, and access ML features across an organization—hiding the complexity of managing such pipelines from end users [48]. For instance, FSs allow end users (e.g., data engineers or scientists) to create features with user-defined pipelines and guarantee no data leakage via a point-in-time join operation [9, 19, 20, 22, 42]. Thus, FSs have emerged as ML-specific database management systems (DBMS) that (a) run pipelines that transform raw data into features, (b) store and manage the features, and (c) serve features for training and inference purposes.

Although FSs offer the functionality mentioned earlier, there is still potential to improve their efficiency from a database systems perspective. One example is ML training, which is an iterative process that involves generating different training datasets with features and labels to train and test an ML model. By recognizing overlap in feature computation across iterations, previously computed features can be reused to generate new features. This can reduce the computational overhead and accelerate the data pipeline for machine learning. Furthermore, by observing users' access patterns, we can propose data source layouts that offer better performance for feature computation pipelines.

We exploit these opportunities in FeathrPO, which is designed to optimize data pipelines for machine learning through the application of optimization techniques rooted in database research [32, 34, 44]. FeathrPO is implemented as an extension to Feathr [13], a state-of-the-art open-source FS widely used in production at LinkedIn that provides native integration with Azure, making it easy to deploy on cloud infrastructure.

Contributions. The goal of this work is to explore some of the untapped opportunities in FSs to enable more effective and efficient data management processes. Our contributions include:

- A characterization of optimization opportunities overlooked in most FS implementations (§3).
- An effective, data layout-aware cost model, primarily designed for pipelines involving point-in-time joins (§4).
- Novel reuse-based optimization techniques for these pipelines (§5).
- A Binary Integer Linear Programming-based data layout selector for determining the optimal global configuration using the proposed cost model (§6).
- Integration of the proposed techniques into FeathrPO, an extension module for Feathr, a popular open-source FS (§7).
- Comprehensive evaluation of the proposed techniques against state-of-the-art baseline FSs, showing significant (up to 3.0×) performance gains on important TPCxAI and Kaggle use cases (§8).

Work done while Rui, Kwanghyun, and Konstantinos were at Microsoft.

Proceedings of the VLDB Endowment, Vol. 16, No. 13 ISSN 2150-8097.
doi:10.14778/3625054.3625060

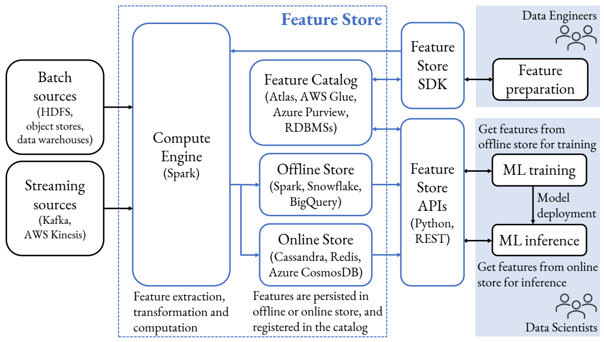

Feature Store
Batch sources (HDFS, object stores, data warehouses)
Streaming sources (Kafka, AWS Kinesis)
Compute Engine (Spark)
Feature Catalog (Atlas, AWS Glue, Azure Purview, RDBMSs)
Offline Store (Spark, Snowflake, BigQuery)
Online Store (Cassandra, Redis, Azure CosmosDB)
Feature Store SDK
Feature Store APIs (Python, REST)
Data Engineers
Feature preparation
Get features from offline store for training
ML training
Model deployment
ML inference
Get features from online store for inference
Data Scientists
Feature extraction, transformation and computation
Features are persisted in offline or online store, and registered in the catalog

Figure 1: Feature store common architecture, including representative examples for its components.

2 BACKGROUND ON FEATURE STORES

FSs allow users to create sophisticated computation pipelines, typically composed of multiple stages, for transforming raw data into relevant features. They are primarily designed to handle *time series* data and offer dedicated APIs and operations that streamline the processing of such data, guaranteeing both point-in-time accuracy and correctness. In this section, we provide a brief overview of the architecture of FSs in §2.1, then describe the widely-used point-in-time join operation in §2.2.

2.1 Feature Store Architecture

Figure 1 depicts the architecture of a FS, including the components that can be commonly found in most existing implementations. Data engineers rely on an SDK library provided by the FS to create pipelines that extract features from raw data by applying operations such as joins or aggregations. These pipelines are executed using a *compute engine* that provides high throughput and flexible behavior, including ingestion from different types of data sources, such as *batch* or *streaming* systems. Since ML *training* and *inference* have different data access requirements, the resulting features are written to two different stores: An *offline store* which is optimized for throughput since training often requires reading large amounts of data; and an *online store* which is optimized for low-latency reads since inference often involves reading a small subset of the data with stringent response time requirements. Data scientists access the features using the FS APIs. Feature definitions are registered into the *feature catalog*, which plays a key role in feature discovery and reuse across users. Finally, FSs often provide additional capabilities such as feature observability, model monitoring, and fine-grained access control.

2.2 Point-In-Time Join

FSs provide a multitude of data transformations and aggregations. However, the *point-in-time join* (also referred to as PIT or ASOF join) [9, 19, 20, 22, 42] stands out as the most crucial operation for generating training datasets. This operation is consistently present in FSs and guarantees that the combined data accurately represents the state of the sources at the specified point in time.

PIT Join Example. Consider an e-commerce site that wants to predict whether a customer will buy a certain item during Labor Week 2022 (see Figure 2). One possible option is to make this prediction based on what a customer bought before and during Labor Week

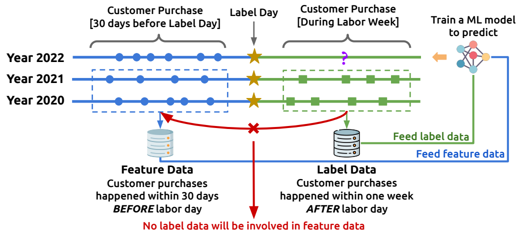
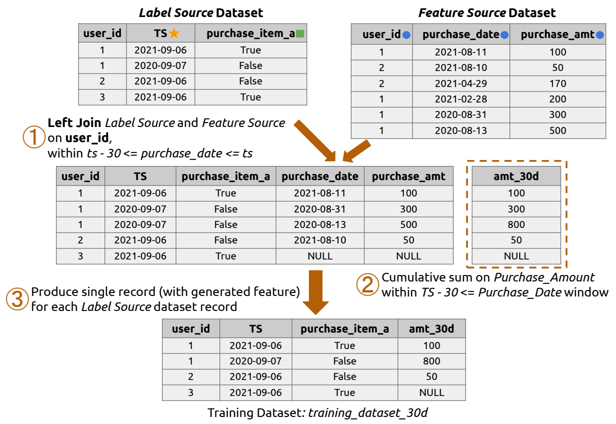

Customer Purchase [30 days before Label Day]
Label Day
Customer Purchase [During Labor Week]
Train a ML model to predict
Year 2022
Year 2021
Year 2020
Feed label data
Feed feature data
Feature Data
Customer purchases happened within 30 days BEFORE labor day
Label Data
Customer purchases happened within one week AFTER labor day
No label data will be involved in feature data

Figure 2: Handling of time series data to predict customer's potential purchases during Labor Week 2022.


① Left Join *Label Source* and *Feature Source* on **user_id**, within *ts - 30 <= purchase_date <= ts*
② Cumulative sum on *Purchase_Amount* within *TS - 30 <= Purchase_Date* window
③ Produce single record (with generated feature) for each *Label Source* dataset record

***Label Source* Dataset**

user_id	TS ★	purchase_item_a 🟩
1	2021-09-06	True
1	2020-09-07	False
2	2021-09-06	False
3	2021-09-06	True

***Feature Source* Dataset**

user_id ●	purchase_date ●	purchase_amt ●
1	2021-08-11	100
2	2021-08-10	50
2	2021-04-29	170
1	2021-02-28	200
1	2020-08-31	300
1	2020-08-13	500

user_id	TS	purchase_item_a	purchase_date	purchase_amt
1	2021-09-06	True	2021-08-11	100
1	2020-09-07	False	2020-08-31	300
1	2020-09-07	False	2020-08-13	500
2	2021-09-06	False	2021-08-10	50
3	2021-09-06	True	NULL	NULL

amt_30d
100
300
800
50
NULL

user_id	TS	purchase_item_a	amt_30d
1	2021-09-06	True	100
1	2020-09-07	False	800
2	2021-09-06	False	50
3	2021-09-06	True	NULL

Training Dataset*: training_dataset_30d*

Figure 3: Example of Point-In-Time join used to compute a window aggregate feature.

in previous years (e.g., 2020 and 2021). Assume we have two data sources, the *label source dataset* containing whether a user bought item *a* in Labor Week 2020 and 2021, and the *feature source dataset* with the purchase amount by each user on a given day, both shown in Figure 3 with sample data. This will be our *label* (or expected prediction) and *feature* data, respectively. These sources need to be combined according to a specific *point-in-time* (or cutoff point) [39]. In addition, window aggregates need to be computed over the combined sources according to some provided time window (e.g., the amount that a customer spent over the 30 days previous to Labor Week). In this context, ensuring point-in-time correctness and avoiding data leakage [47] are critical; failing to do so could result in significant differences in accuracy between training and inference.

The FSs SDKs simplify the process of creating complex feature computation pipelines significantly. For instance, the following code snippet, using the Feathr Python SDK, demonstrates how to create a pipeline for the previous example:

```python
# define PIT join key
join_id = TypedKey(key_column="user_id", key_type="ValueType.INT32")
# define window aggregated feature
agg = WindowAggTransformation(expr="purchase_amt", agg_func="SUM", window="30d")
feature = Feature(name="amt_30d", key=join_id, transform=agg)
# define label source dataset
lsd = HDFSSource(data_path=label_src_dataset, timestamp_column="ts")
# define feature source dataset
fsd = HDFSSource(data_path=feature_src_dataset, timestamp_column="purchase_date")
# define feature based on PIT join
feature = FeatureAnchor("training_dataset_30d", [lsd, fsd], [feature])
```

PIT Join Definition. Next, we use the familiar SQL semantics to define PIT joins. Continuing with the previous example, the PIT join and subsequent window aggregate can be expressed as a correlated subquery, as it is shown in Figure 4a.

```sql
SELECT user_id, ts, purchase_item_a, amt_30d
FROM label_src_dataset l_ds
  LEFT JOIN LATERAL (
    SELECT SUM (purchase_amt)
        OVER (ORDER BY purchase_date ASC) AS amt_30d
    FROM feature_src_dataset f_ds
    WHERE f_ds.user_id = l_ds.user_id
      AND f_ds.purchase_date >= l_ds.ts - 30
      AND f_ds.purchase_date <= l_ds.ts
    ORDER BY purchase_date DESC LIMIT 1) subq ON TRUE;
```

(a) PIT join workflow q_1 with 30 days window aggregate used to compute `training_dataset_30d`.

```sql
SELECT user_id, ts, purchase_item_a, amt_40d
FROM label_src_dataset l_ds
  LEFT JOIN LATERAL (
    SELECT SUM (purchase_amt)
        OVER (ORDER BY purchase_date ASC) AS amt_40d
    FROM feature_src_dataset f_ds
    WHERE f_ds.user_id = l_ds.user_id
      AND f_ds.purchase_date >= l_ds.ts - 40
      AND f_ds.purchase_date <= l_ds.ts
    ORDER BY purchase_date DESC LIMIT 1) subq ON TRUE;
```

(b) PIT join workflow q_2 with 40 days window aggregate.

```sql
SELECT user_id, ts, purchase_item_a,
  CASE
    WHEN amt_30d IS NULL AND amt IS NULL THEN NULL
    ELSE COALESCE(amt_30d, 0) + COALESCE(amt, 0)
  END AS amt_40d
FROM (
F1  SELECT user_id, ts, purchase_item_a, amt_30d, amt
    FROM training_dataset_30d t_ds
      LEFT JOIN LATERAL (
        SELECT SUM (purchase_amt)
            OVER (ORDER BY purchase_date ASC) AS amt
        FROM feature_src_dataset f_ds
        WHERE f_ds.user_id = t_ds.user_id
          AND f_ds.purchase_date >= t_ds.ts - 40
          AND f_ds.purchase_date < t_ds.ts - 30
        ORDER BY purchase_date DESC LIMIT 1) subq ON TRUE
) o_subq;
```

(c) Reuse-based optimized workflow q'_2 using `training_dataset_30d` computed by q_1.

```sql
F2 WITH agg_t_ds AS (
     SELECT MIN (ts) AS min_ts, MAX (ts) AS max_ts
     FROM training_dataset_30d
   )
SELECT user_id, ts, purchase_item_a,
  CASE
    WHEN amt_30d IS NULL AND amt IS NULL THEN NULL
    ELSE COALESCE(amt_30d, 0) + COALESCE(amt, 0)
  END AS amt_40d
FROM (
  SELECT user_id, ts, purchase_item_a, amt_30d, amt
  FROM training_dataset_30d t_ds
    LEFT JOIN LATERAL (
      SELECT SUM (purchase_amt)
          OVER (ORDER BY purchase_date ASC) AS amt
      FROM (
F2      SELECT f_ds.* FROM feature_src_dataset f_ds, agg_t_ds
        WHERE f_ds.purchase_date >= agg_t_ds.min_ts - 40
          AND f_ds.purchase_date < agg_t_ds.max_ts - 30
      ) f_ds
      WHERE f_ds.user_id = t_ds.user_id
        AND f_ds.purchase_date >= t_ds.ts - 40
        AND f_ds.purchase_date < t_ds.ts - 30
      ORDER BY purchase_date DESC LIMIT 1) subq ON TRUE
) o_subq;
```

(d) Optimized workflow q''_2 after introducing semijoin reduction on q'_2.

Figure 4: Example illustrating the rewritings used to compute queries efficiently from previously materialized feature data.

For each record in `label_src_dataset`, the PIT join matches records in `feature_src_dataset` according to the correlated conditions in the `WHERE` clause, and produces a single record per match corresponding to the greatest `purchase_date` that is less than or equal to `ts`. Note that the query preserves all records in the label source dataset in the output even if there are no matches in the feature source dataset (e.g., *user_id=3* in Figure 3). In addition, this PIT join computes a window aggregate feature `amt_30d` containing the sum of purchase amounts over a 30 days window for each user. Observe that the condition `purchase_date≤ts` guarantees PIT correctness, while `purchase_date≥ts-30` is relevant for the window aggregate. Further, it is worth noting that in this particular example, the inner query containing the aggregate could have been defined using a `GROUP BY` clause. However, we opted to use a `SELECT` and `ORDER BY/LIMIT` combination to generalize the PIT join definition because FS pipelines can compute non-aggregate features based on the PIT join result, e.g., through built-in or user-defined functions.

The semantics outlined above pertain to the *left* variant of the PIT join; similar to other SQL joins, the definition can be extended to *inner*, *right*, and *full* variants. Throughout this work, we concentrate on the *left* PIT join, as it is the most frequently utilized in FS implementations. This is due to the fact that non-matching rows can still hold significance during model training, as they may indicate that a selected feature cannot be employed for label prediction.

PIT Join Implementation. The efficient execution of the PIT join can be challenging, and thus, various specialized implementations have been proposed [3]. These implementations often include other operations, such as window aggregation, within a single operator to decrease the number of passes over the same data. An in-depth analysis of PIT join algorithms is beyond the scope of this work; however, a detailed comparison of multiple state-of-the-art algorithms can be found in [49]. In our experimental evaluation (§8.2), we demonstrate that our optimizations are effective and have a substantial impact regardless of the PIT join algorithm selected.

3 OPPORTUNITIES

While FSs are considered a cornerstone to address many of the challenges present in data pipelines for ML, there still exist untapped optimization opportunities for FSs. Next, we discuss a few of them, based on our so-far experience with FSs, and conclude with the scope of this paper:

(O1) *Feature computation reuse.* Feature preparation is an iterative process that involves creating training datasets, evaluating the model's performance, and repeating these steps until desired results are achieved. It is frequent to have overlap computation between the features in each iteration, as well as across features used to train the same or different models. Although FSs provide a unified catalog to register features, it is still left to data engineers to manually identify reutilization opportunities and modify their feature definitions accordingly. Not only is this process error-prone, but it is also not scalable even for a limited number of features—leading to missed opportunities both individually and across organizations.

(O2) *Data sources layout.* FSs create complex pipelines to compute features that frequently involve filtering the original data sources on dimensions such as time. However, they do not optimize the data layout of these sources, which could help to avoid scanning data that is not relevant for certain pipelines execution, and thus, accelerate feature computation and reduce resource consumption.

(O3) *Incremental feature refresh.* Features that have been registered in the FS need to be refreshed as new data is appended to the existing datasets. Even when only a small fraction of the underlying data has changed, FSs recompute features from scratch rather than doing it incrementally, i.e., utilizing their outdated contents and the new data changes to compute an up-to-date version. This is inefficient, leading to longer computation times and higher operational costs.

(O4) *Scheduling of feature maintenance.* Currently, feature refresh is often scheduled based on user-specified time intervals or rather simple policies, neglecting important factors such as feature freshness, refresh schedule of dataset dependencies, and available compute resources. Considering these factors could result in more timely feature updates and more efficient resource utilization.

(O5) *Compute engine selection.* FSs often support multiple engines for computing and writing feature data to both offline and online stores. Although each engine has its own strengths and limitations, including support for different data sources and store connectors, a given deployment typically relies on a single engine for feature computation. Alternatively, the selection of the engine is left to the data engineers on a per-case basis. Automating this selection could improve both performance and overall FS usability.

While some of these opportunities could be tackled within the FS compute engine, we believe that doing it in the FS layer has several advantages. First, FSs may support multiple interchangeable engines, and, using this approach, our techniques can be applicable independent of the engines being used. Second, implementing these techniques within the engine may not be possible, particularly if the FS relies on a non-open source engine. Finally, the FS layer has additional context information (e.g., common plan execution patterns), which allows us to better tailor our optimizations in comparison to a general context approach.

Scope of this paper. In this work, we focus on the offline store of FSs for model training and explore two of the opportunities discussed above in the context of point-in-time join. In particular, we focus on (**O1**) identifying and automatically reusing features computed previously to accelerate the computation of new features (§5), and (**O2**) choosing an efficient data layout for sources taking into account the characteristics of the storage system (§6). We plan to explore other optimization opportunities in the future.

4 COST MODEL FOR DATA PIPELINES

Prior to proposing a cost model for PIT joins, it is important to understand the common operational patterns and recognize the significant impact of data layout on performance. PIT joins primarily operate on time-series data, with both feature source dataset and label source dataset growing over time, and new data arriving naturally in a time (or semi-time) order. The label source dataset is joined with the feature source dataset on non-time columns, but the join result is heavily filtered by predicates on the time dimension, with window aggregation applied afterward. Consequently, the feature data required for the PIT join result is only a subset of the feature source dataset along the time dimension. In addition, the feature data is typically much larger than the label data.

Given these operational patterns, it is evident that the layout of the source datasets on the storage system (e.g., HDFS, object stores, and data warehouses) significantly affects the performance of PIT joins. In particular, *partitioning horizontally* on the time dimension allows the compute engine to skip reading large portions of data, making it a highly effective strategy for feature source dataset layout. Other strategies such as *sharding* (or *bucketing* in HDFS and object stores [55]) both label and feature source datasets could also be used; however, their implementation would require modifying the PIT join implementation. Although the cost model and techniques we present in this paper could be extended to other layouts, we concentrate on horizontal partitioning as it offers the most significant impact for the effort required, leaving exploration of other strategies for future work.

Cost Model. Next, we introduce a cost model for data pipelines in FSs. This model serves two key purposes in our work: (a) facilitating the selection of the best execution plan among alternatives, and (b) guiding the decision on the layout of source datasets.

Assuming a pipeline q, we denote by S_q the set of source datasets read by q. Further, given a source dataset s, we denote by D_s its size, while we use s_p to represent the partition strategy p for s. In our work, we decide to focus on minimizing the data scanned by a pipeline as a proxy to minimize its cost and, thus, define the cost as $C_q = \sum_{s \in S_q} \mathcal{D}_s - \mathcal{U}^q_{s_p}$. We denote by $\mathcal{U}^q_{s_p}$ the benefit brought by s_p to the execution of q. In our case, the benefit is calculated as a weighted sum of three terms, in decreasing order of significance: (a) the size of data in partitions in s that will not be read by q if the

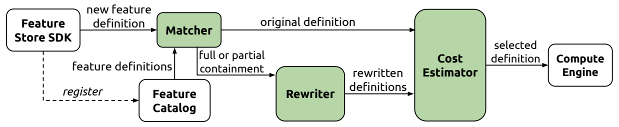


Figure 5: Feature extraction optimization workflow.

partition strategy p is used, (b) the size of data filtered by q once read from the selected partitions, and (c) the #partitions read, i.e., extra partitions add overhead on query planning and scheduling. The weighting coefficients are selected such that less significant terms only have relevance when more significant terms are equal.

Next, we provide details on how we compute the size of these partitions and the selectivity of the filter operations in q. To perform the cardinality estimation while avoiding the excess costs of histogram constructions, we rely on sketch-based quantile estimation. In particular, in this work, we rely on the popular KLL [46] sketches for quantile estimation (i.e., the selectivity of the filter on time is computed using the PDF over the rank domain of the time attribute that KLL encodes). Note that, in our context, we assume append-only datasets—hence, KLL sketches are sufficient. Datasets with updates/deletions and scenarios when we require better error guarantees on the two ends of the underlying rank domain are beyond the scope of this work. However, we note that both concerns, from a sketching perspective, have recently been addressed through REQ [35] and KLL± [61] sketches, respectively.

5 REUSE-BASED FEATURE COMPUTATION

In this section, we extend the FS to automatically reuse existing features to compute newly defined ones. It is important to note that users explicitly specify the features they want to compute and materialize. Thus, our main objective is not to determine which features to materialize ourselves but rather to modify incoming pipelines to leverage previously computed features. Our techniques focus on *time series* data and compute plans containing PIT joins. In addition, we aim at reusing feature results both when there is an exact match between them and when the new features can be incrementally computed using existing ones, e.g., time window overlap between multiple features.

Architecture. Figure 5 depicts our high-level architecture to optimize the feature computation pipelines in FSs; the newly introduced components are shown in green. When data engineers define a new feature using the FS SDK, the feature definition is intercepted by the *Matcher*. This module has access to the FS catalog containing the definitions of already computed features, which we refer to as *materializations*. To guarantee correctness, the *Matcher* needs to evaluate the validity of each materialization definition by checking whether the source datasets used to compute the materialization have been updated since its last computation. Then, for each valid materialization, it evaluates whether the results of the new feature are fully or partially contained in it. For each positive response, the *Rewriter* is invoked with the given pair and the type of match, and it tries to produce an alternative definition to compute the new feature by relying on materialization results. More details about the supported rewritings are provided below. Then, the *Cost Estimator* receives the original feature definition along with any definition produced by the *Rewriter* and chooses the best execution alternative according to the cost function of §4. Finally, the selected alternative is passed on to the compute engine and, after completion, the FS SDK registers the new feature into the catalog.

Reuse-based Computation of Training Datasets. Given a pipeline q_1, our optimization algorithm applies a rewrite rule $q_1 \longrightarrow q_2$ to transform q_1 into a semantically equivalent pipeline q_2. However, q_2 will use any of the previously materialized training datasets. Our algorithm produces rewritings for multiple cases of query equivalence and containment, following the principles of other classic query rewriting algorithms [37, 38, 51, 59].

One particularly interesting scenario appears when one needs to create multiple training datasets through PIT join pipelines using different time windows to tune a model. For instance, assume we want to extend the example shown in Figure 4a, and we would like to explore the ML model accuracy by training it with a time window of 40 days instead of 30 days (Figure 4b). Our algorithm matches q_2 definition with q_1 and produces the rewriting shown in Figure 4c. As it is shown in the SQL fragment *F1*, the rewriting contains two important modifications to the PIT join: (*i*) the outer table is `training_dataset_30d`, the training set generated by q_1, and (*ii*) the filter in the inner subquery is altered to only compute the join over the remaining delta that had not been previously processed. The rewriting also introduces additional expressions applied to the result of `o_subq` to handle `NULL` values correctly. Note that if the time window of the second pipeline were narrower than the first one, such as 10 days, a slightly more complex rewriting adapting well-known incremental view maintenance ideas [40, 41] could be generated. It is also important to emphasize that the proposed rewriting is only valid for the *left* variant of the PIT joins; the exploration of alternative rewritings for other variants of the PIT join is a direction for future research.

After generating the reuse-based optimized plan, we implement further optimizations to enhance the performance of the new plan. Specifically, note that the extent of the speedup is greatly influenced by the acceleration in the inner subquery by computing only the delta over the existing feature. Hence, reducing the amount of data processed in the inner query can improve performance. To that end, as shown in the SQL fragment *F2* in Figure 4d, we compute the minimum and maximum timestamp values in the training dataset, and use those values to filter the scan over the `feature_src_dataset`, which is akin to semi-join reduction [54]. As it will become clear in the following sections, combining these filters with an appropriate partitioning strategy, can result in substantial execution time savings via partition elimination. It is important to note that this semi-join optimization can be applied over the original PIT join plans regardless of the use of the reuse-based optimizations.

6 AUTOMATIC LAYOUT SELECTION

In addition to feature computation reuse, we also introduce a mechanism to optimize the layout of source datasets in FSs. As already discussed in §4, *time series* data is prevalent in FSs, and it is common that feature computation pipelines filter them based on their time dimension. As a result, our proposal focuses on strategies to partition the data horizontally based on their timestamp values.

We state our problem as follows. *Given a workload consisting of pipelines that were executed to compute features in the FS, the collection of source datasets that are read by those pipelines, and the current layout of those source datasets, find a partitioning strategy for each source dataset containing a time dimension such that the overall cost of the workload is minimized subject to a bound on the size of the data that can be rewritten.* To avoid maintaining new copies of the same data and thus increasing the complexity of the ingestion pipelines, throughout this section, we assume that only a single partitioning strategy can be selected for each source dataset. However, note that our solution can be extended to support multiple strategies for the same source, which could bring additional performance benefits to the feature computation process. Additionally, for deploying our solution in a production environment, ensuring data integrity during concurrent writes and data partitioning changes is critical. While we currently use a simple locking scheme, we plan to explore advanced versioning capabilities provided by formats like Delta Lake [1] to simplify our workflow.

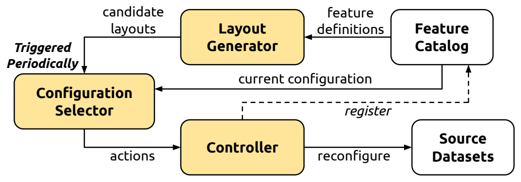


Figure 6: Data layout selection workflow.

Architecture. Figure 6 shows the new modules responsible for automatic layout selection (in yellow color) and their interaction with other FS components. Similar to previous works on index selection [28, 31], *configuration* refers to a certain physical design, i.e., the layouts of a set of source datasets. Currently, our optimization is triggered periodically at a set time interval, which is sufficient with relatively stable workload access patterns. In the future, we may explore automated interval selection or reactive approaches for further optimization. The *configuration selector* requires two inputs: the candidate layouts based on the pipelines executed to compute the features in the FS, and the current configuration of the source datasets read by the pipelines. For the former, we rely on a *layout generator* that retrieves the feature definitions from the FS catalog and use them to generate multiple candidates. Based on its inputs, the *configuration selector* executes a cost-benefit analysis and chooses the best configuration. Finally, the *configuration selector* output consists of a list of actions to apply to the source datasets, which are executed by the *controller*. To prevent the reconfiguration process from becoming a straggler or blocking training and inference jobs for an extended period of time, we place an upper bound on the time spent reconfiguring each data source. The *controller* is also responsible for registering the new data layouts in the catalog. Details about the candidate generation process and configuration selection algorithm are provided next.

Candidate Generation. We retrieve feature definitions from the catalog and extract the source datasets that (*i*) contain a time dimension t, and (*ii*) are filtered by the value in t in the feature definition. Then, for each source dataset, we propose partitioning strategies based on the expression $f(t, e)$: f is a flooring function by granularity e applied on values of t, and e takes a different value (year, month, day, or hour) for each candidate strategy.

Configuration Selection. To formulate our problem, we rely on the notation introduced for our cost function in §4 along with the following notation. Given a workload W, we denote by S_W the

$$\text{Minimize } C_W = \sum_{q \in W} \sum_{s \in S_q} \mathcal{D}_s - \sum_{p \in P_s} x_{sp} \cdot \mathcal{U}^q_{sp}$$

$$\begin{aligned}
\text{subject to} \quad & x_{sp} \in \{0,1\} && \forall s \in S_W, p \in P_s && (1)\\
& \sum_{p \in P_s} x_{sp} = 1 && \forall s \in S_W && (2)\\
& \sum_{s \in S_W} \sum_{p \in P_s} D_s \cdot (1 - X^{t-1}_{sp}) \cdot x_{sp} \le \mathcal{B} && && (3)
\end{aligned}$$

Figure 7: BIP reduction of the selection problem.

set of source datasets read by pipelines in W, which encompasses training sources materialized and read by our optimized pipelines. Further, given a source dataset s, we denote by P_s the set of partition strategies generated for s by the candidate layout generator. For each strategy s_p, we introduce a variable x_{s_p}, denoting whether or not s_p is part of the selected configuration. In addition, we use $X^{t-1}_{s_p}$ to refer to whether s_p is part of the current configuration; this is required to know whether s will actually need to be rewritten if s_p is selected. Finally, $\mathcal{B}$ is our upper bound on the size of the data that can be rewritten. This bound is set depending on several factors, such as the time window for repartitioning and the performance of the compute engine.

We use Binary Integer Programming (BIP), a well-established field of mathematical optimization, to solve the problem of finding the minimum-cost configuration. BIP has been successfully applied to various optimization problems in the field of databases [30, 43, 45, 57]. In a linear programming problem, we aim to minimize the value of an objective function by assigning values to variables, subject to linear inequality constraints. Our problem is stated in BIP terms in Figure 7. Constraint (1) states that each variable x_{s_p} takes values in $\{0, 1\}$. (2) ensures that exactly one partitioning strategy is chosen for a given source (*no partitioning* is a possible strategy), and (3) specifies that the size of the source datasets to be repartitioned cannot exceed the bound $\mathcal{B}$.

Extensive research and development over the years have led to the successful implementation of highly efficient BIP solvers [16, 17]. Our empirical validation using OR-Tools [15] confirms that our solution performs well, with the end-to-end runtime for 10,000 queries and 1,000 source datasets ranging from 6-7s. This includes instantiating the BIP program, which involves loading and proving the data sketches to compute the benefit of each partitioning strategy, as well as solving the BIP itself.

7 IMPLEMENTATION

We have implemented our optimization techniques in FeathrPO, an extension to Feathr [13], a widely-used FS originally created by LinkedIn and recently open-sourced. It provides seamless integration with Azure and other cloud services. Feathr provides a Python interface for accessing all its components, including feature definition and cloud services interaction; the Python client can be easily installed using *pip*. It leverages Spark for feature computation, and provides integration with managed offerings such as Azure Synapse Spark [4] and Databricks Spark [23]. It has its own PIT join implementation [12], and provides a diverse range of connectors for both offline and online stores.

Figure 8 shows FeathrPO's components (*reuse-based optimizer* and *data layout selector*) and their integration with Feathr (gray

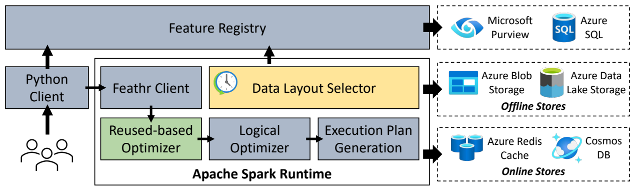


Figure 8: FeathrPO new modules and their integration with other Feathr components (colored in *gray*).

components) on Azure. Below we present the additions and modifications made to integrate FeathrPO within Feathr.

Pipelines Rewriting. The Feathr Python client enables users to create complex feature computation pipelines that can consist of multiple steps, i.e., features that are derived from other features [21]. A feature definition example was shown in §2.2. After the user defines a feature, they can choose to materialize it with an API call, and at that time, the Python client generates multiple files that define the data pipeline to execute, such as label and feature source datasets (Listing 1), feature definitions (Listing 2), etc.

```
"jobName": "training_dataset_30d",
"labelSource": "label_src_dataset"
"featureSource": [{
    "sourceName": "feature_src_dataset",
    "timestampColumn": "purchase_date"
    "timestampFormat": "yyyy-MM-dd"
}]
```

Listing 1: Job Config.

```
{
"featureName": "amt_30d",
"featureDef": "purchase_amount",
"aggregation": "SUM",
"aggWindow": "30d",
"key": ["user_id"]
}
```

Listing 2: Feature Definition.

Subsequently, a Spark program containing references to these files is submitted to the engine. Upon execution, the program instantiates the Feathr client component, which reads the specified files and generates the feature definition instances. Then, the *reuse-based optimizer* in FeathrPO is triggered and, if a new optimized plan that computes a feature based on an existing materialization is selected, it instantiates updated definitions incorporating the details of the optimized plan. Feathr then creates a pipeline to compute these features, optimizes it, and executes the operations within. To support the new feature extraction pipelines introduced in §5, modifications were made to the Feathr logical planner and the component responsible for generating execution plans. Finally, the computed results will be stored in either the offline or online store.

Scheduled Layout Selection. As discussed, Feathr offers integration with managed Spark cloud platforms that support scheduling Spark jobs based on temporal triggers [8, 25]. The *data layout selector* in FeathrPO is executed as a recurring Spark job, with the frequency being configurable. The integer program in the selector is implemented in Java, leveraging the glop solver in OR-Tools [15].

Extensions to Catalog. We extended the catalog to store state information for the operation of FeathrPO. Extensions include (a) registering materialized feature definitions and their data storage path, (b) retrieving feature definitions for reuse-based optimization, (c) registering changes to the layout configuration for a source dataset, (d) retrieving partitioning strategies and source datasets configuration, and (e) collecting and retrieving statistics for a given source dataset.

8 EVALUATION

In this section, we evaluate the benefits of the optimizations in FeathrPO on PIT join pipelines. Our key results are as follows:

Table 1: Summary of dataset statistics (# of rows / size)

Use Case	Label Source (L)	Feature Source (F)	Additional (A)
TPCxAI-UC7 (SF10)	789,225 / 27MB	27,987,766 / 1.5GB	358,818 / 39MB
TPCxAI-UC10 (SF10)	37,696 / 725KB	55,975,921 / 2.1GB	358,818 / 39MB
Favorita	379 / 7.3KB	125,497,041 / 3.9GB	55 / 1.4KB
eCommerce	44,415 / 941KB	104,335,510 / 3.4GB	104,335,510 / 4.1GB

- FeathrPO outperforms the baseline Feathr by up to 3.0×, and FeathrPO optimizations potentially deliver speedups of 1.4–2.5× against the other state-of-the-art PIT join implementations.
- FeathrPO scales better than the baseline Feathr.
- Thanks to our KLL-based cost model, FeathrPO prevents performance regressions across all of our experiments.

Datasets. In PIT join, there are two necessary datasets—feature source dataset (**F**) and label source dataset (**L**). The results of PIT join are often joined with other datasets to generate final datasets for model training. Thus, we have an additional dataset (**A**) in the evaluation to simulate this pipeline. For our evaluation, we use two real-world datasets from Kaggle (i.e., Favorita [7] and eCommerce [10]) and TPCxAI [29, 56] benchmark datasets with different scale factors to form the datasets F, L, A.

Usecase 7 (UC7) in TPCxAI. This use case recommends products to customers based on their shopping history and rating of historically purchased products. L is created by adding an event-timestamp column to the *product rating* table in TPCx-AI benchmark. F is created by joining the *order* and *lineitem* tables. A is the *customer* table. These three tables are joined on the *user ID* column.

Usecase 10 (UC10) in TPCxAI. This use case aims to detect fraudulent financial transactions. L is created by adding an event-timestamp column to the *fraud* table. F is created by extracting relevant columns from the financial *transactions* table. A is the *customer* table. These three tables are joined on the *user ID* column.

Favorita in Kaggle. This dataset is used to predict the unit sales of items sold at different stores during specific holidays. L is created by joining the *holidays events* and *train* datasets. F is created by extracting relevant columns from the *train* dataset. A is the *stores* dataset. These three tables are joined on the *store ID* column.

eCommerce in Kaggle. This use case predicts when an item will be purchased according to the view events on this item in a large online store. L is created by selecting *purchase* events. F is created by collecting all the records of *view* events. Lastly, A is created by extracting the products from *purchase* and *view* events. These three tables are joined on the *product ID* column.

Point-in-time join queries. The PIT join queries that we use over our datasets involve 3-way joins with Label source (**L**), Feature source (**F**), and Additional dataset (**A**), summarized in Table 1.

System setup. For all FeathrPO experiments, we used a cloud setup. We deployed Feathr v0.7.2 [13] on Azure Synapse [6] using Azure Blob Storage [5] as an offline store and Apache Spark v3.1 cluster as an ingestion/compute engine. The Spark cluster runs on YARN with a driver node and 4 worker nodes, each of which has 8 vCores and 64 GB RAM. To demonstrate the applicability of the optimization techniques, we also implemented two other PIT join algorithms. Since there is no cloud deployment on these algorithms, we used a local setup for experiments on them: a system (32GB of DRAM, 10 Intel Xeon P family/Core i7 processors, 500GB SSD) running 64-bit Ubuntu 20.04 and Spark v3.2 using local storage for the offline store.

Reported metrics. For each experiment, we report the mean of the end-to-end execution time of three runs after removing the lowest and highest query times.

8.1 Micro-experiments

In this section, we first evaluate the benefit of individual optimization techniques proposed in FeathrPO. Selecting the right layout for feature source data, particularly for rapid data skipping based on desired date ranges, is crucial to ensure the effectiveness of our rewriting techniques. Depending on the selectivities, we have observed an order of magnitude speedup on the table scan empirically (e.g., 8% selectivity leads to roughly 8.7× speedup on scanning the feature source table with the right partitioning scheme). As a result, all of our experimental results are on properly partitioned feature source data based on the layout selection algorithm in §6.

Impact of reuse-based rewriting and semijoin reduction. Figure 9 depicts the impact of reuse-based rewriting and semijoin reduction using TPCxAI UC7 (SF 10). Reuse-based rewriting itself can harm the end-to-end query performance due to the potentially inefficient shuffling introduced by FeathrPO. This tells us that the optimizations used in FeathrPO are not always beneficial and blindly applying them can give a significant performance regression compared to the baseline Feathr, thereby requiring a proper cost model to prevent such regression. In addition, we empirically observed that the rewritten query, along with semijoin reduction, can increase the coverage of cases when rewritten queries will outperform the baseline due to the savings from data processing cost with the additional filter on the feature source **(F)** dataset.

Impact of feature source (F) dataset selectivity. To further illustrate the importance of cost-based optimization in choosing between rewritten plans and default plans, Figure 10 demonstrates that the performance of rewriting + semijoin reduction mainly depends on the selectivity on the feature source **(F)** dataset. When selectivity is high, it is better to use the default plan for computing features. With cost-based optimization, FeathrPO is able to choose the default plans for 80% and 100% selectivities in this experiment.

Table 2: Cardinality estimates for F' after semijoin reduction from Spark-default and KLL

|F|	|F'|	Spark Default	KLL	Acc (Spark)	Acc (KLL)
27,987,765	1,460,497	1,314,540	1,506,628	90.01%	96.84%
	2,981,515	2,700,783	3,016,420	90.58%	98.83%
	5,288,699	4,875,750	5,303,170	92.19%	99.73%
	10,054,824	9,010,578	10,094,827	89.61%	99.60%

Cardinality estimation with KLL sketches over Spark default. As demonstrated above, the effectiveness of the cost-based optimizer highly depends on the accuracy of cardinality estimates. In FeathrPO, we rely on KLL sketches for cardinality estimation. As Table 2 shows, the KLL sketches consistently outperform the default estimates in Spark.

8.2 End-to-end Evaluation

For this experiment, we mimic the real-world workloads using PIT join queries. Similar to what is described in Figure 4, we are

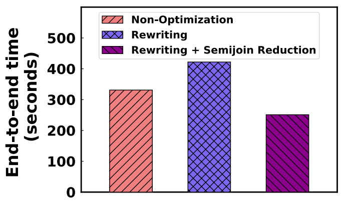


Figure 9: Performance of reuse-based rewriting and semijoin reduction on TPCxAI UC7 (SF10).

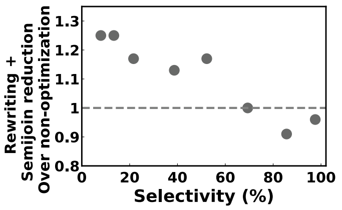


Figure 10: Impact of selectivities on feature source (F) dataset with TPCxAI UC7 (SF10).

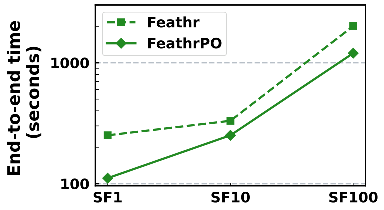


Figure 11: Scalability of FeathrPO and Feathr with increasing TPCxAI UC7 dataset size.

simulating the scenarios when one needs to create multiple training datasets through PIT join pipelines using different time windows to tune an ML model—We extract feature datasets with a time window of 40 days instead of 30 days for TPCxAI UC7, UC10, and Favorita and a time window of 5 days instead of 3 days for eCommerce. We materialize previously computed datasets and reuse them for the follow-up training datasets.

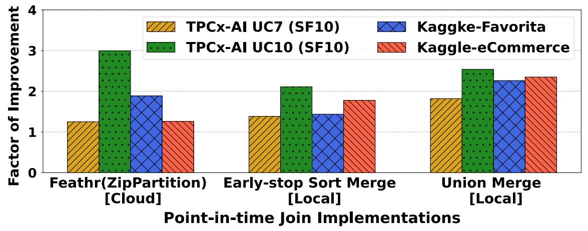


Figure 12: Factor of improvement with FeathrPO on Spark for different datasets and PIT join implementations.

Speedups in different PIT join implementations. We now measure the end-to-end PIT join performance with and without FeathrPO optimizations. Besides the PIT join in Feathr, we also manually implemented FeathrPO optimizations in two other state-of-the-art PIT join algorithms—Early stop sort-merge PIT join and Union PIT join [11, 49]. The factor of improvements with FeathrPO optimizations compared to the baseline (i.e., without FeathrPO optimizations) across four different datasets and four different PIT join algorithms are reported in Figure 12. Our experiments in Feathr are executed in the cloud setup, whereas the other PIT join experiments are performed in our local system setup. FeathrPO outperforms the baseline Feathr across four different datasets by up to 3.0×. FeathrPO optimizations are able to improve the end-to-end query performance even with the other PIT join implementations by up to 2.5×, which proves that FeathrPO optimizations are pluggable to other Feature Stores using different PIT join implementations.

Data scalability. To study the scalability of FeathrPO with increasing dataset size, we used different scale factors of TPCxAI UC7. Figure 11 shows that FeathrPO outperforms the baseline Feathr consistently across three different data scales by 1.3×–2.3×.

9 RELATED WORK

To the best of our knowledge, FeathrPO is the first attempt to optimize feature stores from the perspective of resource efficiency. Thus, we broadly review the related works and position our work.

Point-In-Time Join. The PIT join is fundamental in feature stores but is also used in other data processing systems, such as time-series databases [49]. Custom libraries to improve performance have been proposed [14, 24, 26]. Instead of focusing on improving the implementation of PIT join algorithms, this work applies optimizations without altering the underlying execution engine and can be used for various PIT join implementations, as shown in §8.2.

Feature Engineering Optimization. Previous works have focused on optimizing feature engineering, a vital step in a data pipeline for ML. Columbus [60] is an early work that offers a library of feature selection operations and optimizations. It also examined the performance of its optimizations in the presence of feature overlapping. Additionally, UPLIFT [50] exploits the available parallelism and cache-conscious runtime operations composing featurization pipelines. Our work is complementary to these efforts and such techniques could be adopted by FeathrPO to extend its capabilities. FEDB [18, 33] proposes to use persistent memory to reduce the total cost of ownership of online feature engineering. It adopts an optimization technique called *time window reuse*, which scans the data once for the largest time window and calculates the query result for multiple time windows at the same time. However, three key distinctions between FEDB and our proposed system FeathrPO should be highlighted: (a) FEDB optimizes for online inference with prior knowledge of the windows that will be queried, whereas FeathrPO opportunistically rewrites pipelines without requiring upfront knowledge; (b) FEDB does not support PIT join pipelines, unlike our novel rewriting algorithm; and (c) FEDB focuses on achieving an exact match once aggregate windows are computed, while our techniques can still be used to compute a window incrementally.

Computation Reuse. Materialized views are widely used for computation reuse in traditional databases and data warehouses [34, 38], with various view selection algorithms proposed [2, 28, 44, 58]. Recently, computation reuse has also been applied to data science and machine learning workloads [36, 53]. In this paper, some of our optimization techniques are based on the ideas borrowed from the existing materialized view literature. But by narrowing down the problem scope to the specific context of PIT join, the problem becomes more optimizable than solving a general problem.

10 CONCLUSION

FeathrPO is a new instance of "DBMS for ML" designed to optimize data pipelines for machine learning through the application of techniques rooted in database research. It is implemented as an extension to Feathr, a state-of-the-art open-source feature store widely used in production at LinkedIn. Our results show the optimizations introduced in FeatherPO can provide up to 3× performance gains on use cases such as TPCxAI and Kaggle. We believe that this is just one step forward in the journey of optimizing data pipelines.

REFERENCES

[1] 2019. Delta Lake. https://delta.io/. Accessed: 2023-02-23.

[2] 2022. Amazon Redshift - Automated materialized views. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-auto-mv.html. Accessed: 2022-10-02.

[3] 2022. Apache Spark. https://spark.apache.org/. Accessed: 2022-10-02.

[4] 2022. Apache Spark in Azure Synapse Analytics. https://learn.microsoft.com/azure/synapse-analytics/spark/apache-spark-overview. Accessed: 2022-10-02.

[5] 2022. Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs. Accessed: 2022-10-02.

[6] 2022. Azure Synapse Analytics. https://azure.microsoft.com/en-us/products/synapse-analytics. Accessed: 2022-10-02.

[7] 2022. Corporación Favorita Grocery Sales Forecasting. https://www.kaggle.com/c/favorita-grocery-sales-forecasting. Accessed: 2022-12-20.

[8] 2022. Databricks - Create, run, and manage Databricks Jobs. https://docs.databricks.com/workflows/jobs/jobs.html. Accessed: 2022-10-02.

[9] 2022. Databricks Feature Store - Use time series feature tables with point-in-time support. https://docs.databricks.com/machine-learning/feature-store/time-series.html. Accessed: 2022-10-12.

[10] 2022. eCommerce behavior data from multi category store. https://www.kaggle.com/datasets/mkechinov/ecommerce-behavior-data-from-multi-category-store. Accessed: 2022-12-20.

[11] 2022. Feathr - Point-in-time Correctness and Point-in-time Join. https://github.com/feathr-ai/feathr/blob/main/docs/concepts/point-in-time-join.md. Accessed: 2022-10-02.

[12] 2022. Feathr - Point-in-Time Join Implementation. https://github.com/feathr-ai/feathr/blob/main/feathr-impl/src/main/scala/com/linkedin/feathr/offline/join/DataFrameFeatureJoiner.scala. Accessed: 2022-10-02.

[13] 2022. Feathr: An Enterprise-Grade, High-Performance Feature Store. https://github.com/feathr-ai/feathr. Accessed: 2022-10-02.

[14] 2022. Flint: A Time Series Library for Apache Spark. https://github.com/twosigma/flint. Accessed: 2022-10-02.

[15] 2022. Google OR-Tools. https://developers.google.com/optimization/. Accessed: 2022-10-02.

[16] 2022. Gurobi Optimization. https://www.gurobi.com/. Accessed: 2022-10-02.

[17] 2022. kiwisolver. https://pypi.org/project/kiwisolver/. Accessed: 2022-10-02.

[18] 2022. OpenMLDB. https://github.com/4paradigm/OpenMLDB. Accessed: 2022-10-02.

[19] 2022. Point-in-time Joins in Feast. https://docs.feast.dev/getting-started/concepts/point-in-time-joins. Accessed: 2022-10-10.

[20] 2022. Point-in-time Joins in Feathr. https://feathr-ai.github.io/feathr/concepts/point-in-time-join.html. Accessed: 2022-10-12.

[21] 2022. Point-in-time Joins in Feathr. https://feathr-ai.github.io/feathr/concepts/feature-definition.html. Accessed: 2022-10-12.

[22] 2022. Point-in-time Joins in Hopsworks. https://www.hopsworks.ai/post/a-spark-join-operator-for-point-in-time-correct-joins. Accessed: 2022-10-11.

[23] 2022. Spark on Databricks. https://www.databricks.com/product/spark. Accessed: 2022-10-02.

[24] 2022. Spark PIT: Utility library for Point-in-Time joins in Apache Spark. https://github.com/Ackuq/spark-pit. Accessed: 2022-10-02.

[25] 2022. Synapse Analytics - Integrate with pipelines. https://learn.microsoft.com/azure/synapse-analytics/get-started-pipelines. Accessed: 2022-10-02.

[26] 2022. tempo: Time Series Utilities for Data Teams Using Databricks. https://github.com/databrickslabs/tempo. Accessed: 2022-10-02.

[27] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Godwal, Matteo Interlandi, Alekh Jindal, Konstantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen Zhu. 2020. Cloudy with high chance of DBMS: a 10-year prediction for Enterprise-Grade ML. In *Conference on Innovative Data Systems Research (CIDR)*.

[28] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*. 496–505.

[29] Christoph Brücke, Philipp Härtling, Rodrigo D Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proc. VLDB Endow.* 16, 12 (sep 2023), 3649–3661.

[30] Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, and Soudip Roy Chowdhury. 2016. Reuse-Based Optimization for Pig Latin. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM '16)*. 2215–2220.

[31] Surajit Chaudhuri, Mayur Datar, and Vivek R. Narasayya. 2004. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 16, 11 (2004), 1313–1323.

[32] Surajit Chaudhuri and Vivek Narasayya. 2007. Self-Tuning Database Systems: A Decade of Progress. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. 3–14.

[33] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. 2021. Optimizing An In-memory Database System For AI-powered On-line Decision Augmentation Using Persistent Memory. *Proceedings of VLDB Endowment* 14, 5 (2021), 799–812.

[34] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Foundations and Trends in Databases* 4, 4 (2012), 295–405.

[35] Graham Cormode, Zohar S. Karnin, Edo Liberty, Justin Thaler, and Pavel Veselý. 2021. Relative Error Streaming Quantiles. In *ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems PODS*. 96–108.

[36] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines. In *ACM International Conference on Management of Data (SIGMOD)*. 1962–1976.

[37] Alin Deutsch, Lucian Popa, and Val Tannen. 1999. Physical Data Independence, Constraints, and Optimization with Universal Plans. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB '99)*. 459–470.

[38] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *ACM International Conference on Management of Data (SIGMOD)*. 331–342.

[39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[40] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. *SIGMOD Rec.* 24, 2 (may 1995), 328–339.

[41] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *ACM International Conference on Management of Data (SIGMOD)*. 157–166.

[42] Paul Hargis, Jason MacKay, Raphey Holmes, and Mark Roy. 2021. *Build accurate ML training datasets using point-in-time queries with Amazon SageMaker Feature Store and Apache Spark*. https://aws.amazon.com/blogs/machine-learning/build-accurate-ml-training-datasets-using-point-in-time-queries-with-amazon-sagemaker-feature-store-and-apache-spark/

[43] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11, 7 (mar 2018), 800–812.

[44] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc T. Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *ACM International Conference on Management of Data (SIGMOD)*. 191–203.

[45] Konstantinos Karanasos, Asterios Katsifodimos, and Ioana Manolescu. 2013. Delta: Scalable Data Dissemination under Capacity Constraints. *Proc. VLDB Endow.* 7, 4 (dec 2013), 217–228.

[46] Zohar S. Karnin, Kevin J. Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. 71–78.

[47] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. 2012. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 15:1–15:21.

[48] Laurel J. Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. 2021. Managing ML Pipelines: Feature Stores and the Coming Wave of Embedding Ecosystems. *Proceedings of VLDB Endowment* 14, 12 (2021), 3178–3181.

[49] Axel Pettersson. 2022. *Resource-efficient and fast Point-in-Time joins for Apache Spark: Optimization of time travel operations for the creation of machine learning training datasets*. Master's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[50] Arnab Phani, Lukas Erlbacher, and Matthias Boehm. 2022. UPLIFT: Parallelization Strategies for Feature Transformations in Machine Learning Workloads. *Proceedings of VLDB Endowment* 15, 11 (2022), 2929–2938.

[51] Rachel Pottinger and Alon Y. Halevy. 2001. MiniCon: A scalable algorithm for answering queries using views. *VLDB J.* 10, 2-3 (2001), 182–198.

[52] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. 2015. Hidden Technical Debt in Machine Learning Systems. In *Conference on Neural Information Processing Systems (NeurIPS)*. 2503–2511.

[53] Ahnjae Shin, Joo Seong Jeong, Do Yoon Kim, Soyoung Jung, and Byung-Gon Chun. 2022. Hippo: Sharing Computations in Hyper-Parameter Optimization. *Proceedings of VLDB Endowment* 15, 5 (2022), 1038–1052.

[54] Konrad Stocker, Donald Kossmann, Reinhard Braumandl, and Alfons Kemper. 2001. Integrating Semi-Join-Reducers into State of the Art Query Processors. In *International Conference on Data Engineering (ICDE)*. 575–584.

[55] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering (ICDE 2010)*. 996–1005.

[56] TPC. 2022. TPCx-AI Specification, Version 1.0.2. https://www.tpc.org/tpc_documents_current_versions/pdf/tpcx-ai_v1.0.2.pdf. Accessed: 2022-10-02.

[57] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97)*. 136–145.

[58] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *IEEE International Conference on Data Engineering (ICDE)*. 1501–1512.

[59] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. 2000. Answering Complex SQL Queries Using Automatic Summary Tables. In *ACM International Conference on Management of Data (SIGMOD)*. 105–116.

[60] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization Optimizations for Feature Selection Workloads. *ACM Trans. Database Syst.* 41, 1 (2016), 2:1–2:32.

[61] Fuheng Zhao, Sujaya Maiyya, Ryan Weiner, Divy Agrawal, and Amr El Abbadi. 2021. KLL±: Approximate Quantile Sketches over Dynamic Datasets. *Proceedings of VLDB Endowment* 14, 7 (2021), 1215–1227.