FEBench: A Benchmark for Real-Time Relational Data Feature Extraction

Xuanhe Zhou[*]
Tsinghua University
zhouxuan19@mails.tsinghua.edu.cn

Cheng Chen[*]
4Paradigm Inc.
chencheng@4paradigm.com

Bingsheng He
National University of Singapore
hebs@comp.nus.edu.sg

Mian Lu
4Paradigm Inc.
lumian@4paradigm.com

Qiaosheng Liu
4Paradigm Inc.
liuqiaosheng@4paradigm.com

Wei Huang
4Paradigm Inc.
huangwei@4paradigm.com

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

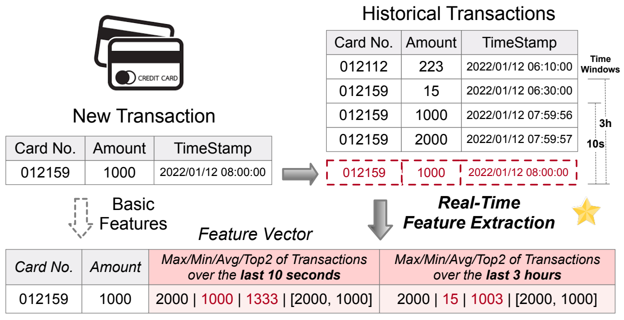
Zhao Zheng
4Paradigm Inc.
zhengzhao@4paradigm.com

ABSTRACT

As the use of online AI inference services continues to rapidly expand in various applications (e.g., fraud detection in banking, product recommendation in e-commerce), systems for real-time feature extraction (RTFE) have been developed to compute features in real-time from incoming data tuples. Also, these RTFE procedures can be expressed using SQL-like languages. However, there is a lack of research on the workload characteristics and benchmarks for RTFE, especially in comparison with existing database workloads and benchmarks (such as TPC-C). In this paper, we conduct a study on the RTFE workload characteristics using massive, open, and realistic datasets (e.g. Kaggle, Tianchi, UCI ML, KiltHub) and practical experience from 4Paradigm. The study highlights the significant differences between RTFE and existing database benchmarks in terms of query structure and complexity. Based on these findings, we have collaborated with our industry partners to develop a real-time feature extraction benchmark named FEBench, which meets the four important criteria for a domain-specific benchmark proposed by Jim Gray. FEBench consists of selected datasets, query templates, and an online request simulator. We use FEBench to evaluate the effectiveness of feature extraction systems and find that each system has its own challenges in terms of overall latency, tail latency, and concurrency performance. FEBench is designed as an open platform, inviting both industry and academia to collaborate on its development and further advancement of RTFE. The project site can be found at *https://github.com/decis-bench/febench*.

1 INTRODUCTION

Online AI applications are rapidly gaining popularity and are expected to dominate the AI market in the near future (e.g., accounting for 44% of the AI market share by 2030 [14]). As a crucial component of these applications, real-time feature extraction (RTFE) aims to timely compute features over the incoming new data tuples. These features play an important role in producing high-quality prediction, often referred to as the *"fuel for AI systems"* [19, 28, 29, 42]. However, with the rise of advanced machine learning techniques (e.g., deep learning), the number of features that need to be computed in real-time has increased significantly (e.g., 662 features for fraud detection [18, 43], 114 features for online recommendation [15, 17, 26], and 458 features for sales forecasting [33]). And RTFE often accounts for a huge proportion of the execution time of the online AI pipeline (e.g., taking 70% time in the sales prediction service of online car purchase platform [1]). To provide a better understanding, we present a few examples of typical RTFE applications below.



Card No.	Amount	TimeStamp
012159	1000	2022/01/12 08:00:00

Card No.	Amount	TimeStamp
012112	223	2022/01/12 06:10:00
012159	15	2022/01/12 06:30:00
012159	1000	2022/01/12 07:59:56
012159	2000	2022/01/12 07:59:57
012159	1000	2022/01/12 08:00:00

Card No.	*Amount*	*Max/Min/Avg/Top2 of Transactions over the **last 10 seconds***	*Max/Min/Avg/Top2 of Transactions over the **last 3 hours***
012159	1000	2000 | 1000 | 1333 | [2000, 1000]	2000 | 15 | 1003 | [2000, 1000]

Figure 1: An example of real-time feature extraction.

Example 1 (Fraud Detection). *For the banking industry, it is crucial to quickly identify fraudulent behaviors, such as multi-location withdrawals, in order to avoid serious financial loss (e.g., the IRS reported a loss of $2.2 billion in a single year [9]). As shown in Figure 1, for a new transaction, if the average transaction amount within 10 seconds (one real-time feature) is much larger than that within the last 3 hours, it may indicate a potential fraud event. In addition to average values, 8 aggregation features (including a customized operator "Top 2") are computed over the two windows to provide a more informative feature vector. This real-time feature extraction process helps to quickly identify suspicious transactions and minimize the potential financial loss.*

Example 2 (Online Recommendation). *In websites like NetFlix, personalized recommendations are provided to users instantly, such as videos displayed on the homepage and recommended videos after a user has finished watching one. To ensure a seamless user experience, it is necessary to immediately update the features for each recommendation, based on a large number of data sources (e.g., historical*

[*]Both authors contributed equally to the paper.

[1]github.com/decis-bench/febench/tree/main/report

```
SELECT * FROM
( SELECT `reqId`,
   `AMT_CREDIT`,
   top_n_frequency(`MONTHS_BALANCE`, 3),
   avg(`amount`) OVER information_0s_3h_100,
   avg(`amount`) OVER information_0s_10s_100, ...
  FROM `information`
   WINDOW information_0s_3h_100 AS (
     PARTITION BY `NAME`
     ORDER BY `eventTime` between 3h
     preceding and 0s preceding MAXSIZE 100),
   information_0s_10s_100 AS (
     PARTITION BY `NAME`
     ORDER BY `eventTime` between 10s
     preceding and 0s preceding MAXSIZE 100) ) AS out0
LAST JOIN
( SELECT `information`.`reqId`,
   `transaction`.`amount`
  FROM
   `information`
   LAST JOIN `transaction` ORDER BY `transaction`.`eventTime`
   ON `information`.`reqId` = `transaction`.`reqId` ) AS out1
ON out0.reqId_1 = out1.reqId_3
LAST JOIN
( SELECT `SK_ID_CURR`,
   distinct_count(`MONTHS_BALANCE`) OVER balance_0_10_
  FROM (SELECT `reqId` AS `SK_ID_CURR` FROM `information`)
   WINDOW balance_0_100_ AS (
     UNION `POS_CASH_balance`
     PARTITION BY `ID_CURR`
     ORDER BY `ingestionTime` rows between 100
     preceding and 0),
   balance_0_10_ AS (
     UNION `POS_CASH_balance`
     PARTITION BY `ID_CURR`
     ORDER BY `ingestionTime` rows between 10
     preceding and 0) ) AS out2
ON out0.reqId_1 = out2.reqId_4;
```

Figure 2: Feature Extraction Query (Fraud Detection)[2].

watches, video information, similar users) and current user actions (e.g., new watches and comments). The real-time feature extraction process is crucial in delivering relevant and accurate recommendations to the user in real-time.

Example 3 (Sales Forecast). *In retail companies like Walmart, accurate prediction of product sales and personalized recommendations to customers are crucial. In this scenario, real-time feature extraction is needed to compute both short-term (e.g., last one hour) and long-term (e.g., last three months) features of user activities, to better understand their purchasing habits and help retailers prepare their products accordingly. Additionally, online sales (e-commerce) require the analysis of different users' preferences based on search (not available in offline sales) and purchase records (e.g., most clicked products in the past 5 minutes, products with the most coupons right now). This is a challenging task, especially when dealing with high-concurrency user requests with ultra-low latency. The real-time feature extraction process plays a crucial role in providing accurate sales predictions and personalized recommendations for e-commerce.*

From above examples, we find that real-time feature extraction is a complex and challenging task that requires (i) the storage of a large volume of incoming data and (ii) the execution of complex operations over multiple, varying-length windows and (iii) the ability to handle high-concurrency query requests. Our research finds that similar applications exist in various business cases from 4Paradigm and our industry partners (e.g., Huawei, 37GAMES, Akulaku, and JD.com). This has prompted the development of various projects (e.g., Flink [20], Feathr [4], FeatHub [3]), which aim to build practical real-time feature extraction systems. These efforts can be broadly categorized into two types of system designs:

(1) General-purpose stream processing systems (e.g., Flink and Storm): Many companies have attempted to construct their feature extraction systems on general-purpose systems like Flink and Storm (e.g., Cloudian, Airwallex, findify AB). These systems possess both batch and stream processing capabilities, making them suitable for online AI inference services (i.e., stream mode for RTFE).

(2) Specialized systems for feature extraction (e.g., OpenMLDB and Tecton): There are two types of industrial-strength systems designed specifically for feature extraction [30, 38, 40]. The first type focuses on serving online features that have been pre-computed during the offline stage. However, it may not be able to produce real-time features with low latency. The second type aims to *update real-time features in online stage*, ensuring the accuracy of the AI systems. For instance, in a fraud detection scenario, features like "whether the user's credit card is locked" must be updated in real-time and cannot rely on offline batch processing that has day-level latency.

A common characteristic of these system designs is that the RTFE procedures can be expressed using SQL like languages, allowing data scientists to focus on describing their feature requirements. As shown in Figure 2, a simplified RTFE query for fraud detection consists of three subqueries. The first subquery employs a single-table window operator to extract the basic profiling information, such as the user's credit and highest monthly balances. The second subquery performs a two-table join to extract information from multiple base tables, such as transaction amounts from the transaction table. The third subquery uses multi-table window operators to calculate new features from two time windows of the POS_CASH_balance table. In the real case, all 668 features from the three subqueries are combined into one feature vector through strict one-to-one join operators [3].

Although real-time feature extraction is increasingly seen as essential for deploying AI models in production, there is currently no research on the workload characteristics and benchmarks for RTFE in comparison with existing database workloads and benchmarks (such as TPC-C). This presents three main challenges. ❶ *How to identify representative RTFE workloads* (**C1**). Due to the massive number of valuable open-sourced realistic datasets available online,

[2]In this paper, the example queries follow the SQL standards of OpenMLDB. Other systems like Flink and Spark Streaming have similar SQL grammars.

[3]https://github.com/decis-bench/febench/blob/main/features/C5

it is laborious to obtain the required datasets (e.g., tabular data with timestamps) and generate the RTFE queries based on the data and task characteristics. ❷ *How to design an effective benchmark with the obtained workloads* (**C2**). It is a non-trivial task to design the benchmark to satisfy the four criteria proposed by Jim Gray [27], which can be contradictory (e.g., adopting more queries may enhance the effectiveness, but negatively affect the benchmark simplicity). ❸ *How to implement the benchmark and gain in-depth insights* (**C3**). Implementing the benchmark in both general-purpose and specialized systems and gaining insights into performance and system designs is also a challenge.

In this paper, we propose a real-time feature extraction (RTFE) benchmark, called FEBench, based on our experience in providing AI solutions for customers from various sectors (including 75 companies in the Fortune Global 500).

What are the key distinctions between RTFE workloads and existing database benchmarks? We analyze the key differences between RTFE workloads (e.g., over 20 real applications from 4Paradigm, over 100 public datasets from popular machine learning repositories) and existing database benchmarks (e.g., transactional [2, 11], analytical [13, 34], and hybrid [12, 23] benchmarks) in consideration of the data distribution, task types, and query structures and complexity (§Section 4, 5).

How can we design an effective and efficient benchmark for real-time feature extraction? We collaborate with industry partners to build a real-time feature extraction benchmark (FEBench). This benchmark is based on a massive number of available RTFE workloads, consisting of selected datasets, query templates, and an online request simulator. We ensured that FEBench meets the four important criteria for a domain-specific benchmark proposed by Jim Gray (§Section 6).

How do we utilize FEBench to compare different existing solutions? We offer a testbed with reusable components (e.g., data loader, workload simulator, performance monitor) to facilitate researchers to design and test RTFE systems with lower overhead on evaluation and implementation. We use FEBench to investigate the effectiveness of feature extraction systems and the preliminary results show all the tested systems have their own problems in different aspects, e.g., (*i*) performance differences can arise due to different implementation techniques: OpenMLDB (a specialized system) runs in assembler code and is significantly faster than Flink (a general-purpose system) that runs in JVM; (*ii*) the long tail problem is more severe in OpenMLDB, which performs poorly in extreme cases (such as the 99th percentile) due to inefficient log writing; and (*iii*) the number of parallel threads has a more significant impact on OpenMLDB than Flink. Our findings reveal that more work is required to improve future feature extraction systems (§Section 7). *We intend to develop FEBench as an open platform to encourage industry and academia to collaborate on the benchmark and further the development of RTFE. The project site is available at https://github.com/decis-bench/febench.*

2 BACKGROUND AND RELATED WORK

In this section, we first introduce the *feature extraction operators* in SQL expressions in Section 2.1. Then we explain how to generate complete *RTFE queries* (the combination of operators that characterize the feature requirements) in Section 2.2. Next, we introduce the *design philosophies and architecture* of feature extraction systems in Section 2.3. Lastly, we discuss why existing database benchmarks cannot be used to evaluate feature extraction systems in Section 2.4.

2.1 Feature Extraction Operators

In real production scenarios, AI models heavily rely on a large number of cross-joined or non-linear features [35] so as to improve their inference capability. Thus, there are various RTFE operators that correspond to different real-time features. Here we showcase five main categories of operator patterns.

(1) Selection + Joins (basic information). Join operators are used to link the tuples of multiple data streams that share common columns. To avoid a large intermediate joined table that can slow down online execution, operators like *last joins* match the tuples in the left stream with the latest matched tuple in the right stream, ordered by timestamp columns.

Example. In Figure 2, the "information" table is joined with the "transaction" table to obtain features such as the historical transaction amounts of a user who has completed the latest transaction, where a sudden increase in the amount values may indicate fraudulent activity.

```
SELECT ‘information’.‘reqId’ as reqId_3,
‘transaction’.‘amount’ as transactionValue
FROM ‘information’ LAST JOIN
‘transaction’ ORDER BY ‘transaction’.‘eventTime’
on ‘information’.‘reqId’ = ‘transaction’.‘reqId’
```

(2) Single Table Windows (basic recent activities). During feature extraction, the time window is a common operator that splits a data stream into buckets of finite sizes (which can be split by different columns), ranks the tuples within each bucket, and performs various aggregations over these buckets. Unlike traditional stream operators, RTFE often concatenates computed features from multiple parts of *the same table with different window sizes* in order to offer features in different time spans.

Example. In Figure 2, the *average amount* features are derived from two windows of the "transaction" table (split by the user name), i.e., "the average amount within 10 seconds" and "the average amount within 3 hours" of the user. This can be expressed as:

```
SELECT AVERAGE(amount_transaction_10s),
        AVERAGE(amount_transaction_3h), . . .
FROM ‘transaction’
WINDOW transaction_3h as (PARTITION BY ‘NAME’ . . .
3h and 0s preceding MAXSIZE 200),
       transaction_10s as (PARTITION BY ‘NAME’ . . .
10s and 0s preceding MAXSIZE 200)
```

(3) Multi-Table Windows (joint recent activities). Similar to traditional joins, multi-table windows enables time windows from different data sources to share common columns (e.g., 8 common columns in the "information" and "POS_CASH_balance" tables). By matching a data tuple with multiple tables, we can compute the time window in each table and concatenate the resulting feature values from the time windows to enrich the final feature vector.

Example. In Figure 2, the "information" and "POS_CASH_balance" tables both contain the "reqId" column, while the "POS_CASH_balance" table contains more profiling information (e.g., credit-card balance, instalment amount). When a new tuple is inserted into "information", we can match the two windowed tables with the new tuple and leverage the output features (on the matched tuples) to enrich online inference information. This function is not limited to OpenMLDB, as similar capabilities can be realized in other systems like point-in-time joins in Feast.

```
SELECT 'reqId',
avg('CNT_INSTALMENT') over POS_CASH_balance_0_100,
FROM ( SELECT 'reqId'  FROM  'information')
WINDOW POS_CASH_balance_0_100 as ( UNION
'POS_CASH_balance'   PARTITION BY 'ID_CURR'   ORDER BY
'ingestionTime' rows between 100 preceding and 0)
```

(4) Table Aggregations. Table aggregations are important in generating non-linear features from columns within a table window. Basic aggregation functions (such as min, max, average) are commonly used, but customized functions for feature extraction such as *top_n_frequency* and *distinct_count* are also useful. For feature extraction, there are five major categories of aggregation functions, including transformation features, accumulated features, preference features, recent activity features, and trend features.

• ***Transformation Features*** are used to convert attribute columns in data sources into the required formats. This can include operations like (i) using the dayofweek(*) function to obtain the day of the week in a timestamp and (ii) using the degrees(*) function to convert radians to degrees. By using transformation features, it is possible to manipulate the data to ensure that it is in the appropriate format for analysis.

• ***Accumulated Features*** are used to obtain accumulated statistics over a period of time. One common way to get accumulated features is by using basic *window+count* operators, such as calculating the total purchase frequencies of products over the last month. This type of feature is useful for understanding trends and patterns over time, such as changes in consumer behavior or product popularity.

• ***Preference Features*** are used to determine the existence and occurrence frequencies of specific items during a period of time. Operators like *window+count_ratio* can be used to achieve this, providing insights into the most frequently occurring activities or items in a given time period. For example, this technique can be used to determine the most frequently purchased products over the last month. By using preference features, it is possible to identify patterns in the data that can be used to guide decision-making and inform future actions.

• ***Recent Activity Features*** are used to reveal changes or distinct values within a recent time period. They can achieve this in two ways: (i) Compute the difference of features in recent tuples with slightly earlier tuples (e.g., last time cycle); (ii) Compute the distinct values within recent tuples (e.g., max/min/sum values of different item families). By using recent activity features, it is possible to better understand the current state of the data and make informed decisions based on these insights.

• ***Trend Features*** are used to reflect the trends in the near future. This can be achieved by using operators like *window+standard_deviation* to compute the occurring distributions of relevant items, such as the average sales in the last week. Trend features reveal *periodic changes in the data and generally involve longer time spans than recent activity features.* By using trend features, we could better understand the long-term trends and changes in the data, allowing for more accurate forecasting and prediction.

(5) Constraints. When performing real-time feature extraction, it is important to avoid having too many tuples within the windows as this can slow down the process. Thus, various constraints can be implemented, such as "maxsize" that limits the maximum number of tuples that can be included within the corresponding window. By setting appropriate constraints, it is possible to balance the need for capturing relevant data while also ensuring that the feature extraction process remains efficient and responsive.

2.2 RTFE Query Generation

Next we introduce how to generate the complete RTFE query. Generally, it is laborious for data scientists to sample candidate features and try out different feature sets. And it is nearly impossible to manually generate the RTFE queries for all the collected datasets. To simplify the process, we proposed to utilize the industry-grade automated machine learning (AutoML) techniques, which can express RTFE in SQL for ease of building AI models [35]. This tool has been implemented in 4Paradigm's commercial product [4] and served in many real-world scenarios (such as Industrial and Commercial Bank of China, UnionPay). Moreover, the data scientists at 4Paradigm have confirmed the effectiveness of the generated queries. As shown in Figure 3, given an AI application and the source data, selecting features and generating corresponding SQL queries involves four steps:

Step 1 (initialization): In the table schema, we first identify the main table (storing the stream data) and secondary tables (e.g., snapshots or read-only tables). Next, we enumerate the relations (corresponding to different RTFE operators) of the main table with all the secondary tables, i.e., for each secondary table T, we enumerate relations between columns of the table T and main table (e.g., one-to-many, many-to-one, many-to-many, slice).

Step 2 (table relation calculator): Next we map the column relations to RTFE operators. As shown in Table 1, for a secondary table, if it has the one-to-one relationship with the main table (e.g., user profiling information), we directly join the secondary table with incoming tuples in the main table and retrieve the whole or a partial set of join results; and if they are one-to-many relationship (e.g., historical transactions of a user), we first join the secondary and main tables and than perform aggregation on the join results. The adopted aggregation functions depend on the secondary table type (e.g., *count* operator for read-only tables, *groupby_count* operator for appendable tables). Each mapped operator pattern corresponds to a candidate feature (represented by an output column).

Step 3 (feature enumerator): After extracting all the candidate features, we iteratively generate useful feature sets in two steps: (i) *Feature set generation:* We employ beam search to explore proper

[4]https://github.com/4paradigm/autox

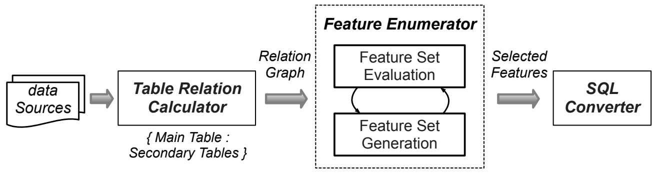


Figure 3: The workflow of RTFE query generation

Table 1: Table relationships and mapped operators. *Events* indicate recent activities; *Status* describes general properties.

Table Type	Relationship	Mapped Operators
read-only table	one-to-one	last join
	one-to-many	aggregation + left join
appendable table	one-to-many	aggregation (events)
	many-to-many	aggregation + union
snapshot table	one-to-one	last join
	one-to-many	aggregation (status)

combinations of candidate features [?]. That is, we select the most promising feature to expand based on the metrics like AUC [24], which measures the prediction quality of the downstream machine learning model; (ii) *Feature set evaluation:* We evaluate the candidate feature sets according to the task target (e.g., the AUC metric) and select the best solution as the new set of features. This iterative procedure is terminated once a specified condition (e.g., the maximum iteration time) is met.

Step 4 (SQL converter): Finally, we convert the selected features into an semantic-equivalent SQL query, and implement the SQL query into the feature extraction system (e.g., the "DEPLOY" command in OpenMLDB). The generated SQL queries for real datasets are available at https://github.com/decis-bench/febench.

2.3 Systems for Real-Time Feature Extraction

Real-time feature extraction refers to *on-demand feature computation and request response in the online stage.* Although there are various products that support feature extraction (e.g., *Michelangelo* in Uber [38], *Zipline* in Airbnb [21], *Feathr* in Microsoft [4], *OpenMLDB* in 4Paradigm [7]), some systems pre-compute the feature values in offline and store in caching for online requests, which is not in the scope of this paper. As shown in Figure 4, a real-time feature extraction system typically has three main modules:

(1) Online feature extraction is essential for real-time processing of new data streams into feature values that enable timely predictions by models. The online storage is memory-based, containing only the latest feature values to model the current state of the world. Online stores support multiple copies of table data to ensure high availability. With the RTFE query deployed in advance (Section 2.2), the systems use an optimized query engine to process online requests. Different from traditional stream systems, they enhance the procedure from multiple aspects, such as supporting (i) double-layered skiplist to efficiently compute TopN operators over time windows and (ii) overlapped window reuse to enhance the data requests [19, 22] (see the example execution plan in Figure 14).

(2) Offline feature extraction is commonly used to persist feature data over extended periods (often months or years), and conduct batch model training with these data. The feature data is usually stored in data warehouses or data lakes (e.g., Hadoop, BigQuery,

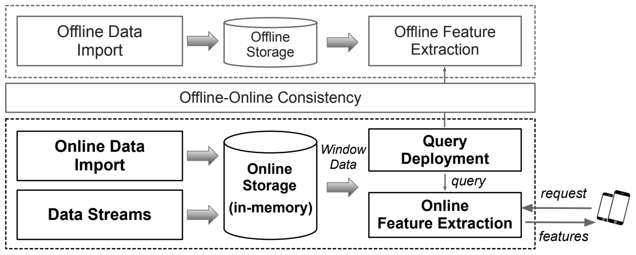


Figure 4: The general architecture of RTFE systems.

Snowflake, Redshift). Offline feature extraction shares the same feature extraction query as the online part.

(3) Offline-and-online consistency. The machine learning model requires a consistent view of features across development (offline batch training) and production (online inference). Subtle differences in the features can cause significant changes in the prediction outcome. For example, Varo, an online bank from the US, discovered that inconsistent execution definitions of "account balance" between offline and online stages cause significant model quality degradation at production [6]. They use the account balance from yesterday at offline and the current account balance at online, which caused inconsistency. Therefore, maintaining a consistent view of feature definitions across offline and online feature extraction is essential for an industry-grade RTFE system.

2.4 Existing Benchmarks

System benchmarking is a highly active area of both research and industry communities [16, 41]. And most standard benchmarks are derived from real data and typical queries, including transactional benchmarks [2, 11], analytical benchmarks [13, 34], and hybrid transactional/analytical benchmarks [12, 23, 31]. In this section, we introduce these benchmarks and explain the challenges of evaluating real-time feature extraction (Table 3).

2.4.1 *Transactional Benchmarks.*

Online transaction processing (OLTP) benchmarks evaluate the ability to maintain business data and process high-concurrency transactions (involving limited tuples).

Scenarios. The scenarios of OLTP benchmarks involve two critical considerations. First, since the data stored in OLTP systems is generally critical to the business, it is vital to ensure the atomicity, consistency, isolation and durability (ACID) of the data. Second, OLTP systems must efficiently handle high-concurrency transactions with short response time (e.g., within milliseconds).

Example. TPC-C [11] simulates a real transactional scenario, where a company (with multiple warehouses and sales districts) processes client orders. TPC-C provides a write-heavy workload, which contains 92% write operators over 9 tables under default settings [25]. TPC-C tables can be scaled to different sizes, indexed based on the number of configured warehouses. The benchmark metric is throughput (e.g., tpmC), which reflects the efficiency of processing concurrent simple operators. However, as shown in Table 3, *TPC-C does not support stream processing; and the operator patterns in the query are relatively simple*, e.g., with single table access and no joins/unions of multiple tables. Besides, RTFE often supports *appending operations* of incoming data, rather than updates in TPC-C.

Table 2: Operator Comparison. The operators with underlines do not exist in TPC-DS/TPC-H.

	Top 10 most frequently used operators
FEBench	*last_join*; *avg*; *max*; *distinct_count*; *top_ratio*; *min*; *order_by*; *window (union)*; *partition_by*; *group_by*
TPC-DS	*sum; case; count; distinct; with; order_by; left_join; limit; group_by; or*
TPC-H	*group_by; order_by; count; left_join; avg; case; having; with; exists; min*

Table 3: Benchmark Comparison. Note second-level latency is acceptable in stream benchmark (non-hard real time).

	Workload	**Real-Time**	**Time-Series**	**#Aggregations**	**#Windows**	**#Subqueries**	**#Joins**
TPC-C [11]	transaction	strong	×	$0 \sim 2$	×	×	×
Sysbench [2]	transaction	strong	×	×	×	×	×
TPC-H [13]	analytics	weak	×	$0 \sim 8$	×	$0 \sim 4$	$0 \sim 8$
JOB [34]	analytics	weak	×	$1 \sim 6$	×	×	$0 \sim 17$
CH-benCHmark [23]	hybrid	medium	×	$0 \sim 8$	×	$0 \sim 4$	$0 \sim 8$
StreamBench [46]	streaming	medium+	✓	$0 \sim 2$	~ 1	×	$0 \sim 2$
FEBench	RTFE	strong	✓	$5 \sim 800$	$0 \sim 10$	$1 \sim 18$	$0 \sim 14$

2.4.2 Analytical Benchmarks. Online analytical (OLAP) benchmarks evaluate the performance of complex data analysis tasks
Scenarios. Different from OLTP systems, OLAP systems aim to efficiently process large-scale table scans, aggregations, data joins from multiple tables, and perform multi-dimensional operators (e.g., with up to three level subqueries).
Example. TPC-H [13] simulates a real scenario, where a wholesale supplier delivers goods worldwide. The workload contains 22 business queries, each of which performs complex data operators (e.g., joins, subqueries). TPC-H does not consider write operators, and *the dataset size remains constant* during workload execution. The benchmark metrics include both throughput (e.g., QphH) and total execution latency. Besides, TPC-DS is a more complex OLAP benchmark than TPC-H, with 99 queries that include structures like table unions that do not exist in TPC-H queries. These OLAP benchmarks only test batch processing capacity over global data, and *do not consider online processing for data streams.* Thus, RTFE systems require *a new benchmark to evaluate real-time complex analytics over data streams.*

2.4.3 HTAP Benchmarks. Hybrid transactional/analytical processing (HTAP) benchmarks aim to efficiently support both operational workloads (e.g., small transactions with high update ratios) and analytical workloads (e.g., with complex access patterns) within the same system.
Scenarios. First, HTAP systems perform real-time data analytics between online transactions. Second, they need to prevent the interference of analytical queries over the dynamically-changing data tables.
Example. CH-benCHmark [23] provides a mixed workload based on TPC-C and TPC-H benchmarks. It enables separate serving of transactional and analytical queries by two types of clients. It merges the two schemas into a single one to allow analytical queries to access transaction tables. However, for feature extraction, HTAP benchmarks face the similar challenges of OLAP benchmarks, such as the lack of support for time-series data and the need of executing analytical queries for relatively long time (in batch mode).

In summary, existing benchmarks are not suitable for designing domain-specific real-time feature extraction (RTFE) benchmarks, because they are significantly different from the example RTFE workloads in Section 1 (e.g., only support SPJ query or lack online testing). To design a domain-specific RTFE benchmark, we need to solve three main problems: (i) how to collect suitable datasets and queries that simulate the real stream data (Section 4); (ii) how to identify the critical differences between RTFE and existing database workloads (Section 5); (iii) how to select representative workloads (e.g., with diversified operators and computation complexity) for a domain-specific RTFE benchmark (Section 6).

3 BENCHMARK OVERVIEW

3.1 Design Goals

We design the feature extraction benchmark by following the *4 benchmark design criteria* proposed by Jim Gray [27].

Relevance. The benchmark covers a wide range of feature extraction behaviors, including different operator complexities. We have collected over 100 real feature extraction workloads from various sources, including *45 Kaggle datasets [10], 11 Tianchi datasets [8], 8 KiltHub datasets [5], 28 UCI ML datasets [1], and 26 applications in 4Paradigm.* These datasets cover the major feature extraction operators (Section 2.1) and scenarios that heavily rely on real-time feature extraction, such as ride prediction, healthcare, energy consumption, sales forecast, and fraud detection.

Simplicity. The benchmark is designed to eliminate redundant tests and be easily understandable. First, we apply clustering techniques to the collected RTFE datasets and only use a small part as query templates, which represent typical RTFE oeprator patterns and reduce redundant tests on similar datasets. Besides, for each query template, we separately describe the data distributions, semantics, and operator patterns for ease of understanding.

Portability. The benchmark is applicable to different feature extraction systems that support SQL-like language. The query templates are written in SQL expressions. With minor modifications (e.g., replacing the customized function with the combination of basic operators), these query templates can be easily migrated to a new feature extraction system.

Scalability. The benchmark includes real datasets of different data sizes and distributions, allowing it to simulate various incoming data sizes and changing patterns (Figure 6). Besides, each dataset

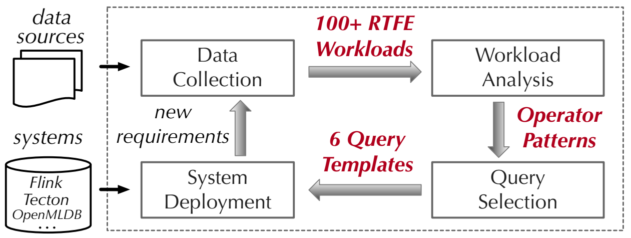


Figure 5: The workflow of FEBench generation.

can be scaled to larger or smaller sizes by following their real distribution of incoming data.

3.2 Benchmark Methodology Overview

Based on the design goals, we build the benchmark in four steps.

Workload Collection. This module includes two parts. First, we search for datasets in various public AI repositories (e.g., Kaggle, Tianchi, UCI ML, and KiltHub) to cover as many AI applications as possible. We extract datasets that meet the real-time feature extraction settings, such as (i) being in tabular data format and (ii) having at least one timestamp column and (iii) being large enough to support minutes of tests. Besides, we obtain 26 real datasets from 4Paradigm [22]. For each collected dataset, we use either a manual-crafted RTFE query by data scientists or the output query of an automatic query generation tool (Section 2.2). Note the output queries also require manual verification (e.g., checking for any unreasonable features) and often achieve much higher execution efficiency than manual-crafted queries.

Workload Analysis. We compare the collected datasets and queries with existing database benchmarks. We analyze the data distributions (e.g., table schemas, incoming data distributions) and query structure differences (e.g., the types of supported operators, the number of complex operators). We then summarize the major data and query characteristics and task targets in feature extraction.

Workload Selection. To ensure the simplicity and effectiveness of the benchmark, we cluster the RTFE queries into templates that combine queries with similar operator patterns and scenario requirements. First, we rank the importance of different operators (e.g., the relation with execution latency) with logistic regression and assign each operator an importance weight. Second, we then use the weighted operators as the query feature vector and apply DBSCAN [32] to divide the origin queries into clusters, each denoting a typical feature extraction scenario (a query template in our benchmark). Note that, to balance between the benchmark simplicity and effectiveness, we tune the DBSCAN parameters (e.g., *eps* controls the minimal distance of datasets within the same cluster, *min_samples* controls the minimal datasets in the same cluster) and obtain 6 clusters with relatively few outliers. Besides, we try to pick datasets that come from different scenarios around the centroid of each cluster to better cover the diversified feature extraction cases. In FEBench, there are six workloads from five typical AI applications, each with various operator patterns.

Deployment on Target Systems. After selecting the workloads (i.e., query templates and real datasets), we implement them on appropriate systems, such as general-purpose systems like Flink [20] and specialized systems like OpenMLDB [22]. Under the same

Table 4: Histogram information for the 118 datasets.

Tables	1	2	3	4	5	6	7	8	10
#Dataset	29	10	13	21	13	22	3	5	2

Data Size	0-10GB	10-20GB	20-50GB	50-100GB	>100GB
#Dataset	62	26	15	9	6

Table 5: The selected datasets from 118 real datasets.

Cluster	Task	Tables	Columns	Rows
C0	Ride Prediction	1	11	2.62×10^6
C1	COVID Forecast	1	6	3.54×10^6
C2	Energy Forecast	8	61	8.0×10^6
C3	Sales Forecast	7	85	1.5×10^{10}
C4	Loan Evaluation	9	245	1.0×10^9
C5	Fraud Detection	10	773	1.3×10^{11}

benchmarking environment, we test the performance of these systems with the selected workloads and obtain some interesting findings (e.g., the trade-off between execution efficiency and system compatibility) by profiling the execution results.

Workflow. The workflow involves five steps (see Figure 5). Firstly, we extract suitable workloads from a variety of public and industry-grade data sources, which involve over one thousand of different ML datasets and take manual efforts to filter unsuitable datasets. Secondly, we compare the collected workloads with other benchmarks, highlighting the unique characteristics of the RTFE workloads. Thirdly, to reduce the evaluation overhead, we cluster origin queries and select six representative query templates. Finally, we explain how to implement the benchmark on feature extraction systems and evaluate their performance from different perspectives.

4 WORKLOAD COLLECTION

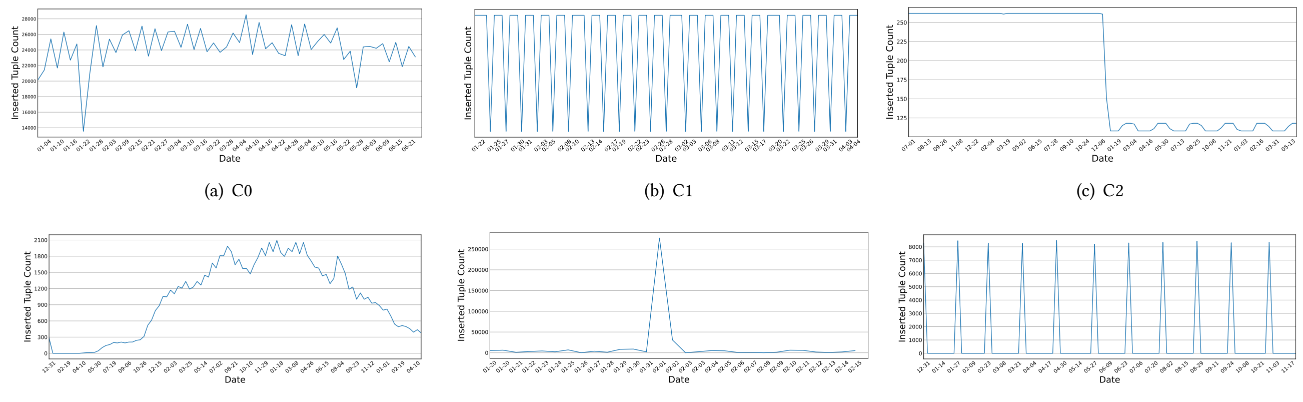
In this section, we explain how to collect and prepare the RTFE workloads (including the datasets and feature extraction queries).

Finding 1. *The selected 118 workloads cover 5 common feature extraction scenarios, comprising relational data with various timestamp distribution (e.g., cycles, sudden bursts). In these workloads, the number of tables is within [1,10], and the data sizes span from 2MB to 10TB (Table 4).*

We spent over ***2 person years*** to comprehensively analyze over 1000 machine learning tasks from multiple popular open data sources (e.g., Kaggle [10], Tianchi [8], UCI ML [1], KiltHub [5]). For example, *Kaggle* is one of the largest online communities of data scientists and ML practitioners, with 579 competitions that provide real decision-making tasks and datasets (last checked at February 14, 2023). For real-time feature extraction, any selected dataset must meet the following requirements:

(1) Relational data: Most online decision-making tasks store data in tabular format, where each tuple represents an instance and each column corresponds to a basic feature;

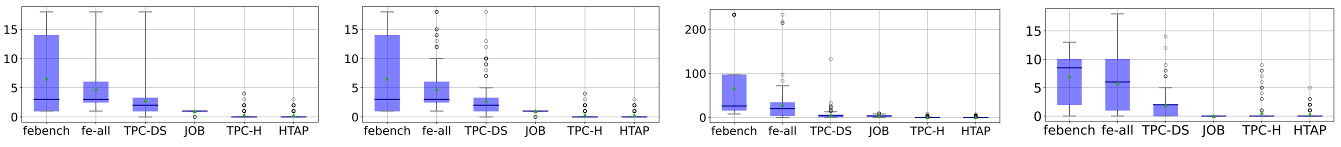
(2) Timestamp feature: Real-time feature extraction requires the updating of computed features by the most recent data. Thus, any selected dataset must contain a "timestamp" feature that simulates the various incoming data distributions in real online scenarios (e.g., periodic cycles for the flu forecast task in Figure 6 (b), random bursts for the loan payment task in Figure 6 (e));



(a) C0 (b) C1 (c) C2

(d) C3 (e) C4 (f) C5

Figure 6: The real data distribution of selected six datasets in FEBench.



(a) Join (b) Selection (c) Aggregation (d) Group/Order

Figure 7: Contrast of query operators (y-axis denotes the number of corresponding operators). Note FEBench denotes the selected 6 RTFE query templates, FE-all denotes the whole set of 118 RTFE queries, and TPC-DS/JOB/TPC-H are analytical queries.

(3) Data scales: We need a dataset that includes at least one table with timestamps (usually the main table) and containing over 1×10^6 tuples. This allows us to simulate the real-world data incoming scenarios and test the performance for minutes. Note, similar to other benchmarks like TPC-C and Sysbench, we only insert tuples by the order of their timestamps during evaluation.

Based on these requirements, we have collected 118 datasets, including *26 internal datasets from 4Paradigm, 45 Kaggle datasets, 11 Tianchi datasets, 8 KiltHub datasets, and 28 UCI ML datasets.* For example, from the over 500 Kaggle competitions, we first exclude non-tabular datasets and examine each tabular dataset to ensure it has a "timestamp" column and meet our data distribution criteria (e.g., the average interval between two tuples is no longer than 1 minute). This yields 45 potentially useful Kaggle datasets for real-time feature extraction. For each collected dataset, we generate the RTFE query for the collected datasets (see Section 2.2).

These RTFE workloads have three characteristics. First, *the collected datasets cover a wide range of data distributions.* As shown in Table 4, the number of tables ranges from 1 to 10, and the dataset sizes span from 1MB to 10TB. Second, the operator patterns in RTFE queries are affected by the datasets. For example, for datasets with a single table (typically used for model training), their RTFE queries involve multi-table windows (of different sizes) over the same tables; for datasets with multiple tables (e.g., 2-6 tables), their queries may contain tricky subqueries of dozens of levels (e.g., joining multiple tables as the intermediate results). Third, *the sizes of most datasets are no larger than 50GB*, because (i) RTFE qeuries are generally executed over most recent data (other data are stored in HDFS) and (ii) only the data tuples within window operations are required. Note we also have relatively large datasets (e.g., over 10TB), whose RTFE queries involve complex structures (e.g., joining the incoming tuple with static tables) and do not involve window operations.

5 WORKLOAD ANALYSIS

With the collected workloads, next we demonstrate our analysis of the unique workload characteristics of RTFE in comparison with the state-of-the-art database benchmarks.

5.1 Observations

Based on the discussion in Section 2.4, we further analyze the detailed operator distributions of RTFE workloads (FEBench) with transaction (TPC-C), analytic (TPC-DS, TPC-H, JOB), and hybrid workloads. And results are shown in Figure 7.

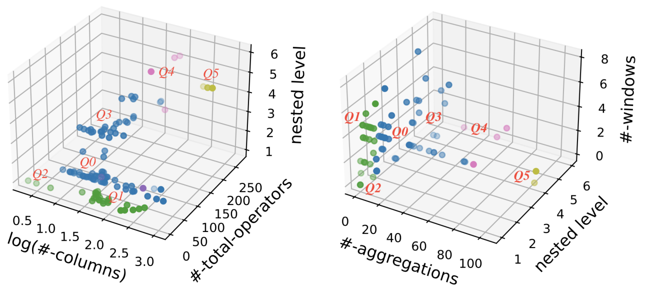
Finding 2. *Among the 118 queries, they support 15 typical operators and various customized aggregations. Different from analytical queries, most of the RTFE queries involve even more complex query structures over table windows.*

RTFE Operators vs Transactional Operators. First, both RTFE and OLTP workloads support high-concurrency queries. However, OLTP supports data update, while RTFE only supports appending data to the end of data streams. Second, OLTP only involves simple queries (e.g., single table scans), while RTFE contains queries with complex operators, and needs to process these operators in time windows. Third, OLTP systems stress the ACID characteristics and are generally disk-based, while systems that support RTFE queries focus on the high efficiency of processing complex operators and adopt in-memory architectures.

RTFE Operators vs Analytical Operators. As shown in Figure 7, RTFE queries cover a much larger range of operator space compared with OLAP queries. To quantify, the maximum query (#-operators, #-aggregations observed is (477, 38) for TPC-H, (883, 73) for TPC-DS

Table 6: The statistics of selected query templates

	features	keywords (SQL)	aggregations	1-1 joins	time windows	subqueries	window unions	max window size
Q0	40	46	11	0	2	2	0	200
Q1	as	49	25	0	10	10	0	200
Q2	45	62	45	8	1	2	0	no limit
Q3	117	156	88	7	9	9	7	100
Q4	262	382	162	9	1	18	12	100
Q5	679	681	652	10	1	3	0	no limit



(a) High-level features (b) Highest-weighted features

Figure 8: Query Clustering Analysis. The 118 origin RTFE queries is divided into 6 clusters (query templates).

and (4969, 321) for RTFE. Besides, most of the RTFE queries have a distinct operator distribution from TPC-DS/TPC-H, e.g., some RTFE queries have hundreds of aggregations while the TPC-DS queries have 40 aggregations at most, and many of the RTFE aggregations do not exist in TPC-DS queries (e.g., topN, count_where, and string joins). These observations imply that RTFE is richer in query operators and has much more complex operator patterns than TPC-DS/TPC-H. Note that analytical queries in HTAP benchmarks [23] have similar characteristics as OLAP queries, and so also cannot well handle the RTFE cases.

Summary. Compared with existing databases queries, feature extraction queries bring new challenges: (i) They involve some time-consuming operators (e.g., *orderby over long table windows*, *window unions*), which are uncommon cases in traditional database queries (Table 2); (ii) The query structures correspond to a large number of real-time features and are very complex, e.g., hundreds of aggregations over multiple data streams (Table 3); (iii) They require to handle high-concurrency requests and ensure strict real-time guarantees (e.g., the latency of dozens of seconds are intolerable) for many online AI-driven applications like fraud detection.

6 BENCHMARK GENERATION

In this section, we first explain how to generate the FEBench benchmark by selecting templates out of the collected RTFE workloads, some of which may contain similar query operators/structures, such that affecting the benchmark portability. Next we provide scenario analysis of these templates.

6.1 Query Template Selection

With the 118 collected datasets and feature extraction queries, we select representative templates based on both the clustering results and scenario characteristics (Figure 8).

Specifically, we utilize the clustering algorithm *DBSCAN* to divide the 118 generated RTFE queries based on their feature vectors. First, for each RTFE query, the feature vector is composed of five parts: (i) the number of output columns, which reflects the result scales; (ii) the total number of query operators, which reflects overall query complexity; (iii) the occurrence frequencies of complex operators (e.g., joins, windows, set unions), which reflect the detailed operator-level complexity; (iv) the highest level of nested subqueries (nested level), which reflects the query-structure complexity; (v) the constraints of maximal tuples in windows.

Since query features have different importance to the evaluation effectiveness (e.g., time windows are often more crucial than simple aggregations), before query clustering, we need to evaluate the relation of these features with the execution characteristics. That is, we train a logistic regression model, with the above five kinds of features as input and the execution time of a RTFE query as output. Then we utilize the regression weights of each feature as their clustering weights (e.g., 4.518 for nested level, 15.037 for last joins, 16.132 for windows). Note different from traditional database queries, RTFE queries generally involve time windows, and so the query complexity is not directly affected by the batch data scale.

Next, with these weighted query features, we utilize a density-based spatial clustering algorithm *DBSCAN* to divide these origin queries. There are five main steps. (i) Choose an epsilon distance and a minimum number (e.g., 3 queries) to define a dense region. (ii) Randomly select an unvisited query and check its neighborhood within epsilon distance. If the number of queries in the neighborhood is greater than or equal to the minimum number, create a new cluster and add the queries and its neighbors to the cluster. If the query is not in a dense region, mark it as noise (we iteratively adjust the epsilon distance value so as to ensure there is no query marked as noise) and move to the next query. (iii) Repeat the process until all queries have been visited. The benefit of *DBSCAN* is that it does not need to manually set the cluster number and can better reflect the query distribution.

Finally, the clustering results are shown in Figure 8. We find that the 118 queries are divided into 6 clusters. In each cluster, we extract queries around the centroid as the candidate templates. Besides, since AI applications have different feature extraction requirements, around the centroid of each cluster, we try to pick queries that come from different scenarios to better cover the feature extraction cases. *As a result, we have picked 6 query templates from the 118 RTFE queries, where 1 for traffic, 1 for healthcare, 1 for energy, 1 for sales, and 2 for financial transactions.*

6.2 Query Template Analysis

Finding 3. *The selected 6 query templates have similar operator patterns as the 118 RTFE queries. These templates cover 6 main RTFE tasks and have diversified operator patterns, which are relevant to the datasets (e.g., table relations) and task types.*

With the selected 6 workloads (denoted as *C0-C5*), we explain why they follow the four benchmark criteria (Section 2.3) by analyzing the data and query characteristics of each workload (Table 6). Note, for each workload Ci, we denote the query template as Qi.

6.2.1 Ride Duration Prediction (C0). The task aims to predict the total ride duration of taxi trips in New York City. The task is highly relevant to the real-time traffic monitoring, which involves the massive GPS data of taxis and buses.

Data (6 months). For the ride-duration-prediction task, we have collected 6 months (ranging from 2016-01-01 00:00:17 to 2016-06-30 23:59:39) of open NYC transportation data. Generally, there are at least 20,000 new records each day. Besides, the arrival time of the records is relatively random (ranging from 12,000 to over 28,000). For the data schema, it only contains one relation table with 11 columns, including basic features like pickup time, geo-coordinates, and the number of passengers.

Query (single table window). The query ($Q0$) involves both basic features (e.g., *pickup_datetime*, *trip_duration*) and windowed features (e.g., distinct count over the *pickup_latitude* column of stream data in 1 hour). Since there is only one base table, $Q0$ only covers fundamental RTFE operators (e.g., 11 aggregations), and windows of multiple data partitions of the same table.

6.2.2 COVID19 Forecast (C1). The task aims to predict the cumulative number of confirmed COVID19 cases in various places all around the world, as well as the number of resulting fatalities in the near future.

Data (2.5 months). For the COVID-forecast task, we have collected around 2.5 months (ranging from 2020-01-22 00:00:00 to 2020-04-07 00:00:00) of covid19 data. First, the temporal data is periodically inserted per 3 days. Second, in each cycle, the peak value is similar (around 310) and lasts for 2 days. In the data schema, it also contains one base table, which includes limited features (6 columns) like the regions, confirmed cases, and fatalities at different time periods. Note C1 is not hard real-time and is taken as a special test case.

Query (multi-table windows). First, different from $Q0$, the query ($Q1$) involves more window operators, e.g., there are 25 window-aggregations operators in $Q1$, while $Q0$ only has 11. The reason is that $Q1$ contains much fewer features (6 table columns) and it is vital to derive effective new features to support accurate forecast. Second, $Q1$ involves different window sizes. For example, the death cases in last one day or one month may both be useful in different COVID phases. And so $Q1$ can help to test the capability of multi-table window processing [20].

6.2.3 Wind Power Forecasting (C2). The task aims to predict the hourly power generation at 7 wind farms 48 hours in advance.

Data (3 years). For the wind-power-forecast task, we have collected around 3 years (ranging from 2009-07-01 00:00:00 to 2012-06-26 12:00:00) of open wind-power data. Until Dec. 6th 2011, the number of incoming temporal data steadily keeps around 265 every 48 hours. After that, the incoming data sharply drops and periodically changes around 120. In the schema, it contains 10 relation tables together with 61 columns.

Query (windowed join over multiple tables). In the query ($Q2$), to extract the joint features from multiple tables (e.g., features of nearby farms and recent hours), it conducts multi-joins on 8 tables, which can well test the online joining performance. Interestingly, $Q2$ mainly extracts temporal features (e.g., the last distinct values of timestamp features), which is different from other queries that involve a large number of aggregations on integer columns. Note that $Q2$ does not conduct window operators, and so all the records may contribute to the feature computation and model inference, which is completely different from the traditional stream cases.

6.2.4 Sales Prediction (C3). The task aims to predict the sales trend and replenish goods intelligently for a casual wear retailer. The sales can be significantly affected by various factors like locations, seasons, product sources, and even weather.

Data (3.5 years). For the sales-prediction task, we have collected 3.5 years (ranging from 2017-12-31 16:00:00 to 2021-05-30 16:00:00) of real data in Uniqlo. Before May 30th 2018, data is rarely inserted into the database. After that, the incoming data quickly increases and, from Feb 3rd 2019 to Aug 4th 2020, the number of new incoming data is over 1200 each day. And after Aug 4th 2020, the incoming data slightly decreases (over 300 tuples each day). In the schema, it owns 7 relation tables together with 85 columns.

Query (window unions, multi-level subqueries). For the query ($Q3$), it involves 113 RTFE operators and 15GB batch data, most of which are aggregations over joined window tables. Besides, different from above templates ($Q0-Q2$), (i) $Q3$ performs set conjunction operators (e.g., unions) over two data streams, since multiple tables in $Q3$ have some of the same feature columns; (ii) $Q3$ has 6-level subqueries at most and involves both base and windowed tables, which are costly to process. Thus, $Q3$ can help to test the complex subquery processing performance.

6.2.5 Untrustworthy Prediction (C4). The task aims to predict whether customers will pay back their loans on time for a credit card company. C4 owns 9 tables together with 1GB data and 245 columns, most of which occur in two days and cause sudden bursts. In the query ($Q4$), it involves 110 basic features and 122 aggregations. Since it needs to characterize the customer behaviors, there are *multiple window-count operators* to reflect the recent activities of the customers. Besides, it has *17 subqueries*, which involve base tables, single-table windows, or multi-table windows. Thus, $Q4$ is a bit more complex than $Q3$ in the operator patterns, and the relatively large intermediate table sizes in C4 can significantly affect the processing efficiency.

6.2.6 Fraud Detection (C5). Fraud detection is a particular case of the outlier detection problems, which aims to estimate the chance of fraudulent activities within milliseconds. Here we consider a fraud detection case in banking scenario. We have collected 11 months (ranging from 2017-12-31 16:00:00 to 2018-11-30 16:00:00) of real data in a big bank. First, the temporal data is periodically inserted per week. Second, in each cycle, the peak value is similar (around 8000). C5 owns 10 tables with over 13GB base data and 773 columns. The query ($Q5$) contains *the most query operators* among the 6 query templates. It involves 659 RTFE operators, most of which are multi_last_value directly taken from the results of multi-table joins. Since $Q5$ does not involve windows, it utilizes *the whole data (all the past user behaviors)* to compute features for incoming stream data.

Summary. The benchmark meets the four criteria of a domain specific benchmark by Jim Gray (Section 2.3). Specifically, the 6 selected query templates own diversified operator patterns. Overall, the first 3 templates are from the public datasets, which have relatively simple operator patterns, but are much more complex than traditional analytic queries (e.g., involving dozens of aggregations or many multi-table joins/windows). And the rest 3 templates are from the real applications in 4Paradigm, which reflect more tricky computation paradigm and are hard to ensure ultra-low latency. Each template has unique operator patterns (e.g., $Q0$ and $Q2$ involve completely different operators, and $Q3$ performs much more window operators than $Q5$). We will show their superiority in the evaluations.

7 EVALUATION OF DIFFERENT SYSTEMS

In this section, we introduce how to implement FEBench. As a starting point, we utilize FEBench to compare the general-purpose system (Flink [20]) and specialized system (OpenMLDB [22]). [5]

7.1 Overview of FEBench Pipeline

As shown in Figure 9, FEBench consists of three components: data loader, workload simulator and performance monitor. Taking fraud detection as an example, *data loader* is responsible for loading static data such as the basic information of both users and shops into the system; *workload simulator* preloads all of the historical transactions into DRAM, and then sends the transaction one by one to simulate the arrival of a new transaction. With each transaction, the system performs a predefined online feature extraction query and records the new transaction data. Performance monitor writes down the response time of each transaction and reports performance metrics such as 50th/99th/999th percentile latency [39] (denoted as TP-50/TP-99/TP-999). For simplicity, we also define tail latency as TP-99/TP-999. FEBench is implemented based on JMH (Java Microbenchmark Harness) [37], which communicates with different system through their Java client.

7.2 Experiment Setting

Hardware Setup We deploy both the systems and FEBench on a server with 40 Cores 2.2 GHz Xeon(R) E5-2630 (2 sockets, 640KB/2.5MB/25MB for L1/L2/L3 caches of each socket, respectively), 500 GB, and we use 7.3 TB hard disk as storage. The OS is CentOS-7.9 with kernel 3.10.0.

System Setup We test two typical systems that can support feature extraction in SQL-like languages. First, Flink is a popular general-purpose platform for real-time feature extraction. The version of Flink is 1.15.2, and we deploy one job manager node and three task

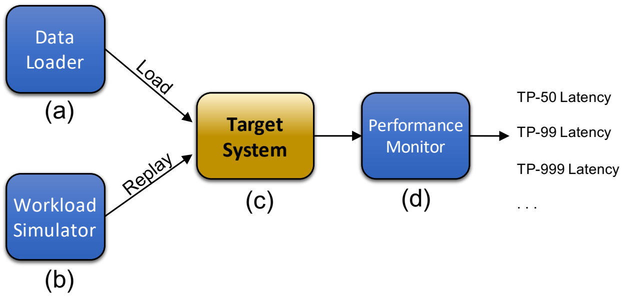


Figure 9: The overview of FEBench pipeline.

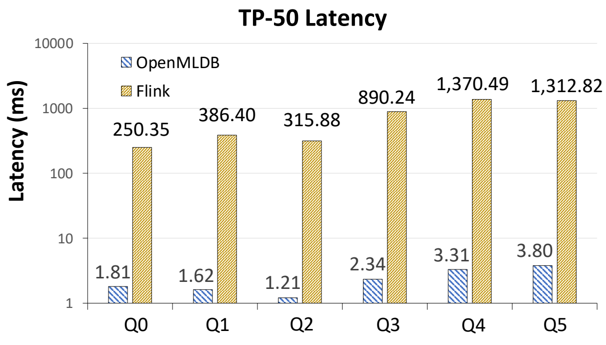


Figure 10: The TP-50 latency of all tasks.

manager nodes as the Flink cluster. Flink caches part of the data in DRAM (the watermark mechanism is well-tuned to ensure all the online required data is cached in memory) and persists data in RocksDB engine. Second, OpenMLDB is a popular specialized platform for real-time feature extraction. The version of OpenMLDB is 0.6.4, and we deploy three tablet servers and one name server as an OpenMLDB cluster. OpenMLDB stores all the data in DRAM. To keep data consistent, OpenMLDB writes logs on HDD for newly added transactions.

Experiment Outline FEBench performs Q0 to Q5 on both OpenMLDB and Flink, and we first display the preliminary results including TP-50 latency and the tail latency. We also study the impact of increasing working threads on both two systems. After that, we compare the execution plans of different systems for the same query to understand the choices made in their designs.

7.3 Preliminary Results

In the following, we present some preliminary results and observations. *Our main goal is not to compare the end-to-end performance of these two systems. Instead, we demonstrate the usage of* FEBench *for showing the impact of different implementation and design from both systems.*

Observation 1: Different choices of implementation techniques can result in large performance gaps. As shown in Figure 10, FEBench reports the TP-50 latency for all tasks. The TP-50 latency of Q0, Q1, and Q2 is shorter than in Q3, Q4, and Q5 because the latter queries require more complex operators. Meanwhile, Flink is almost two orders of magnitude slower than OpenMLDB in all

[5] Some vendors are also evaluating their systems (e.g., Feathr) with FEBench. The experimental results will be continuously updated at https://github.com/decis-bench/febench

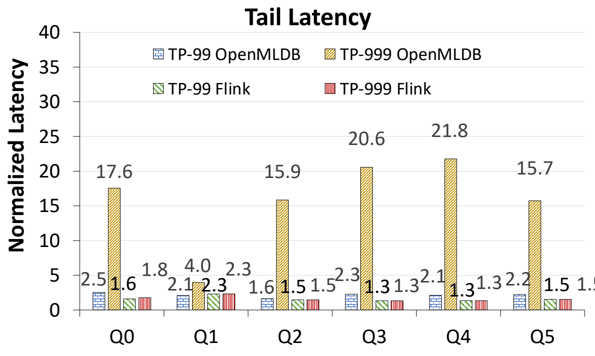


Figure 11: Normalized tail latency (All values normalized to that of the TP-50 latency for each system).

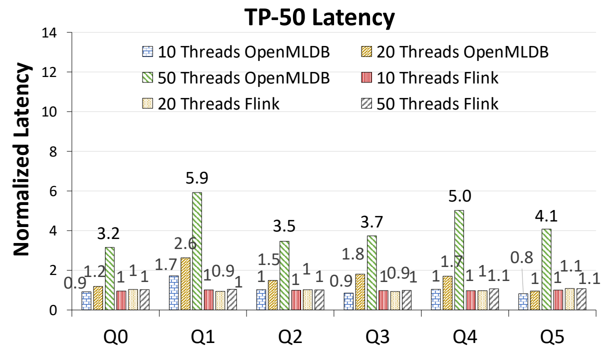


Figure 13: Normalized TP-50 latency (All values normalized to that of the 5 threads).

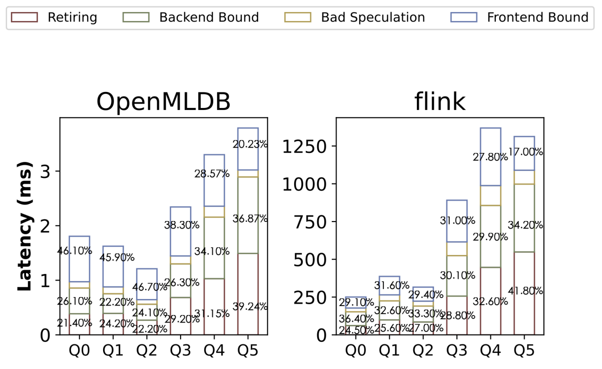


Figure 12: Micro-Architectural Metric Analysis

workloads. Performance differences are primarily caused by implementation choices. To support more general-purpose application scenarios and cross-platform deployment, Flink is implemented by using Java, and all RTFE queries are executed in JVM. The OpenMLDB was originally designed for latency-sensitive applications (e.g. financial anti-fraud). Like other high-performance in-memory databases such as MemSQL [36], OpenMLDB utilizes Low-Level Virtual Machine (LLVM) [44] to transform the given RTFE queries into assembler code, which is much faster than that computed in JVM [45].

Observation 2: OpenMLDB has an obvious long tail issue, while Flink's tail latency is more stable. Figure 11 shows that OpenMLDB's TP-999 latency increases rapidly compared with TP-99 latency in all tasks. To study the growth rate of tail latency, we normalize all TP-999, TP-99 latency to that of TP-50 latency (e.g, the Qx-TP99 and Qx-TP999 latency of DBx are normalized to the Qx-TP50 latency of DBx). In particular, the TP-999 latency is up to 10.32× that of TP-99 latency, and is up to 21.8× that of TP-50 latency from Q0 to Q5 tasks. Such long-tail issues have been observed on other low-latency in-memory databases [22]. In contrast, Flink's tail latency is much more stable. As shown in Figure 11, the TP-999 latency only increases up to 12.69% compared with TP-99, and only increases up to 2.3× compared with TP-50. Long-tail problems are often encountered in low-latency in-memory databases [22]. OpenMLDB stores both the historical transactions and the newly added transactions in the same data storage engine, and OpenMLDB writes logs to the hard disk after a new transaction is inserted. The long tail latency is mainly caused by the back-end `fsync` operations on the persistent storage triggered by the database log write. Flink stores the newly added transaction data in a third-party database, and the execution of RTFE and new data insertion occur in different data engines. Furthermore, as discussed in observation 1, Flink takes a longer time to complete each RTFE query which in effect decreases the frequency of new data insertions, thus reducing the performance impact of database log writes.

Observation 3: The increase in the number of working threads has a less impact on Flink. All values in Figure 13 are normalized to that of five threads (e.g., the latency of Qx-10 threads DBx, Qx-20 threads DBx, and Qx-50 threads DBx are normalized to the latency of Qx-5 threads DBx). As shown in the left part of Figure 13, OpenMLDB's TP-50 latency does not raise rapidly when the threads number increases from five to 20. However, when the number of working threads is set to 50, the TP-50 latency increases significantly (raise up to 5.92× to that of five threads). Flink has smaller change in TP-50 latency when the number of working threads increases.

7.4 Finer-Grained Analysis

To further analyze the performance bottlenecks of RTFE systems in finer granularity. We decompose the execution latency (in cycles) into the metrics specified in Intel's Top-down Micro-architecture Analysis Method (TMAM), including retiring (refers to the time of executing instructions), bad speculation (refers to the time wasted due to branch prediction errors), backend_bound (refers to instructions that cannot be dispatched due to resource shortages), frontend_bound (refers to the time of fetching instructions and decoding them into executable micro-instructions).

As shown in Figure 12, for Q0-Q2 in OpenMLDB, the most time-consuming metric is frontend_bound (over 45%). However, for Q3-Q5 in OpenMLDB, the proportion of frontend_bound gradually decreases (20.23%-38.3%), the most time-consuming metrics are backend_bound (26.30%-36.87%) and retiring (29.20%-39.24%). The time ratio of bad_speculation remains stable for Q0-Q5.

Specifically, we have three observations.

First, in OpenMLDB, when a user's feature extraction request arrives, the system switches to the feature extraction instruction stream. Therefore, there are many instruction stream switches in FEBench, which causes instruction cache-misses and TLB-misses

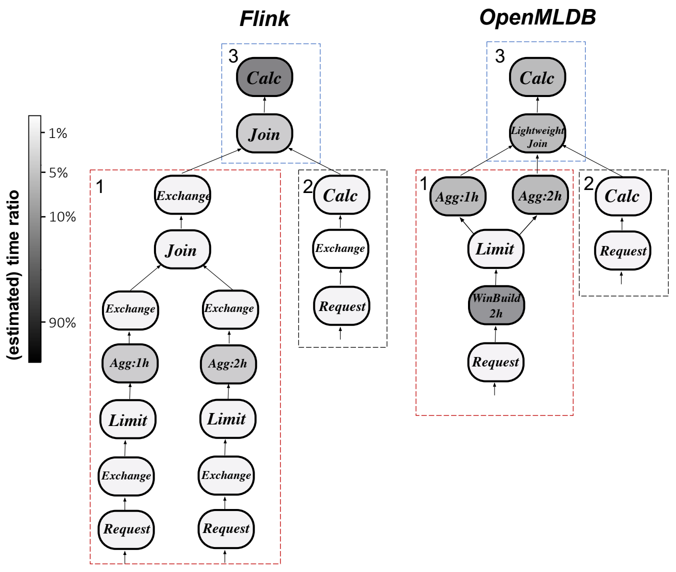


Figure 14: Example Execution Plans (Q0).

(frontend_bound). For Q0-Q2, because the RTFE query is straightforward, frontend_bound becomes the bottleneck of the program. In Q3-Q5, frontend_bound decreased, and the reason will be explained below.

Second, when there are many data cache-misses, backend_bound will be higher. Compared to Q0-Q2, Q3-Q5 have more sub-tables and more RTFE operators and subqueries (as explained in Section 6.2), resulting in more intermediate calculation results. This increase in data cache-misses caused backend_bound to rise.

Third, retiring represents the completion of instruction execution. A higher Retiring indicates a higher IPC (instructions per cycle) for the program. As analyzed earlier, the feature computation operations in Q3-Q5 are most complex, for which more instructions are executed and retiring has a higher proportion in the entire execution process.

7.5 Example Execution Plan

In this part, we use Q0 as an example to show the difference in execution plans for the same query. As described in Section 6, the target of Q0 is to predict the ride duration of taxi X. Except extracting some basic information of taxi X, Q0 needs to access the historical ride data of taxi X in the last one hour and the last two hours, and executes several aggregation operators on these two time windows. As shown in Figure 14, the execution plan consists of three stages (marked as 1, 2, 3 in the figure): 1) extract two time windows of taxi X, and execute aggregations, 2) extract basic information of taxi X, and execute calculations between different select columns, 3) join the results of the first two parts, and execute calculations and output results. As shown in Figure 14, a series of operators have been defined.

(1) **Request**: Locate the required data.

(2) **Exchange**: Pass the intermediate results to other nodes.

(3) **Agg:1h/Agg:2h**: Aggregate on 1-hour/2-hour time window.

(4) **Limit**: Truncate data.

(5) **Join**: Put the intermediate results together.

(6) **Calc**: Calculate on the data.

We have the following two observations on the execution plans.

Observation 4: To reduce the latency of each RTFE query, OpenMLDB has made several optimizations at the execution plan level. Q0 requests to read two table windows (past one hour and two hours of taxi X). The default behavior of the execution plan is to read the data of two table windows separately (shown on the left side of Figure 14). OpenMLDB senses the overlap of the table windows and reads only the data of the larger 2-hour table window, and performs the aggregation operator on the 1-hour time window and 2-hour table window respectively, which reduces the duplicate data reading overhead (shown on the right). Meanwhile, OpenMLDB proposes a customized operator to optimize the latency of the join operator. To reduce the data copy overhead during the join operator, OpenMLDB proposes a lightweight join operator, which only joins the index of the data.

Observation 5: The execution plan of Flink is designed to handle larger data sets and the possibility of collaboration between multiple nodes. When the data set is huge, the data of RTFE query may be distributed on different nodes. The time window data fetching, aggregation, basic information extraction, computation, etc. may result in data transfer across nodes. Flink takes this potential data transfer into account by adding an exchange operator. The purpose of the exchange operator is to check whether the result of previous operator needs to be carried across nodes. As a result, Flink is more suitable for running on multi-nodes with large data sets.

8 CONCLUSION AND FUTURE WORK

Real-time feature extraction is an emerging trend and widely taken as a necessity to take the AI applications into production. In this paper, we first explain how to borrow the ideas in relational data and SQL to conduct feature extraction for real-world applications. Next, based on 100+ real applications, we have proposed a benchmarking architecture FEBench for real-time feature extraction, involving data collection, query analysis, query selection, and system deployment. The preliminary results show that FEBench can effectively reflect the pros and cons of both the general-purpose system (Flink) and specialized system (OpenMLDB) and facilitate the development of real-time feature extraction systems.

In the future, we are targeting at building an open-source benchmark community for AI-driven decision-making applications from three aspects. First, the benchmark is open. Given the methodology, if our partners contribute their workloads, then we could repeat the methodology and generate the updated benchmarks. Second, the testbed is open. Industry partners could use our testbed to evaluate their own systems. Third, the experimental comparison is open. The industry partners could submit their performance results, and our website maintains the ranking of each system.

REFERENCES

[1] https://archive.ics.uci.edu/ml/index.php.
[2] https://github.com/akopytov/sysbench.
[3] https://github.com/alibaba/feathub.
[4] https://github.com/feathr-ai/feathr.
[5] https://kilthub.cmu.edu/.
[6] https://medium.com/engineering-varo/feature-store-challenges-and-considerations-d1d59c070634.
[7] https://openmldb.ai/.
[8] https://tianchi.aliyun.com/.
[9] https://www.irs.gov/pub/irs-prior/p3415–2021.pdf.
[10] https://www.kaggle.com/competitions.
[11] https://www.tpc.org.
[12] https://www.tpc.org/tpcds/.
[13] https://www.tpc.org/tpch/.
[14] Forecast: The business value of artificial intelligence. In *Gartner*, 2018.
[15] S. P. Anderson. Advertising on the internet. *The Oxford handbook of the digital economy*, pages 355–396, 2012.
[16] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.
[17] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-based systems*, 46:109–132, 2013.
[18] R. J. Bolton and D. J. Hand. Statistical fraud detection: A review. *Statistical science*, 17(3):235–255, 2002.
[19] J. Cai, J. Luo, S. Wang, and S. Yang. Feature selection in machine learning: A new perspective. *Neurocomputing*, 300:70–79, 2018.
[20] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.
[21] S. Charrington. Machine learning platforms.
[22] C. Chen, J. Yang, M. Lu, and et al. Optimizing in-memory database engine for ai-powered on-line decision augmentation using persistent memory. *Proceedings of the VLDB Endowment*, 14(5):799–812, 2021.
[23] R. L. Cole, F. Funke, L. Giakoumakis, W. Guy, and et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, page 8. ACM, 2011.
[24] E. R. DeLong, D. M. DeLong, and D. L. Clarke-Pearson. Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. *Biometrics*, pages 837–845, 1988.
[25] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013.
[26] D. S. Evans. The online advertising industry: Economics, evolution, and privacy. *Journal of economic perspectives*, 23(3):37–60, 2009.
[27] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (1st Edition)*. Morgan Kaufmann, 1991.
[28] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.
[29] M. A. Hall and L. A. Smith. Practical feature subset selection for machine learning. 1998.
[30] S. Hur and J. Kim. A survey on feature store. *Electronics and Telecommunications Trends*, 36(2):65–74, 2021.
[31] G. Kang, L. Wang, W. Gao, F. Tang, and J. Zhan. Olxpbench: Real-time, semantically consistent, and domain-specific are essential in benchmarking, designing, and implementing htap systems. *arXiv preprint arXiv:2203.16095*, 2022.
[32] K. Khan, S. U. Rehman, K. Aziz, S. Fong, and S. Sarasvady. Dbscan: Past, present and future. In *The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014)*, pages 232–238. IEEE, 2014.
[33] I. Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.
[34] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.
[35] Y. Luo, M. Wang, H. Zhou, Q. Yao, W.-W. Tu, Y. Chen, W. Dai, and Q. Yang. Autocross: Automatic feature crossing for tabular data in real-world applications. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1936–1945, 2019.
[36] MemSQL, 2013. https://www.memsql.com/, Last accessed on 2022-01-26.
[37] OpenJDK, 2013. https://openjdk.java.net/projects/code-tools/jmh/, Last accessed on 2020-11-15.
[38] L. Orr, A. Sanyal, X. Ling, K. Goel, and M. Leszczynski. Managing ml pipelines: feature stores and the coming wave of embedding ecosystems. *arXiv preprint arXiv:2108.05053*, 2021.
[39] T. percentile. Tp-x. https://support.huaweicloud.com/intl/en-us/productdesc-apm/apm_06_0002.html, 2019.
[40] C. Sun, N. Azari, and C. Turakhia. Gallery: A machine learning model management system at uber. In *EDBT*, pages 474–485, 2020.
[41] Y. Tay. Data generation for application-specific benchmarking. *Proceedings of the VLDB Endowment*, 4(12):1470–1473, 2011.
[42] T. Tsai. Competitive landscape: Ai startups in china. In *Technical Report*.
[43] S. Wang. A comprehensive survey of data mining-based accounting-fraud detection research. In *2010 International Conference on Intelligent Computation Technology and Automation*, volume 1, pages 50–53. IEEE, 2010.
[44] Wikipedia. *LLVM*, 2019. [Online; accessed 02-July-2022].
[45] S. Zeuch, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyzing efficient stream processing on modern hardware. *Proceedings of the VLDB Endowment*, 12(5):516–530, 2019.
[46] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 659–670. IEEE Computer Society, 2017.