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 online AI inference services have been rapidly deployed in many emerging applications such as fraud detection and recommendation, we have witnessed various systems developed for real-time feature extraction (RTFE) to compute real-time features over incoming new data tuples. Also, the RTFE procedures can be expressed in SQL like languages. However, there is not any study about the workload characteristics and benchmarks for RTFE, specifically in comparison with existing database workloads and benchmarks (such as TPC-C). In this paper, we start with a study on the RTFE workload characteristics from massive open realistic datasets (e.g., Kaggle, Tianchi, UCI ML, KiltHub) and the practice experience in 4Paradigm. The study demonstrates significant differences of RTFE to existing database benchmarks in terms of query structures and complexity. Based on those RTFE workloads, we have cooperated with our industry partners and built 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 online request simulator. We utilize FEBench to investigate the effectiveness of feature extraction systems and find all the tested systems have their own problems in different aspects (e.g., overall latency, tail latency, and concurrency performance). FEBench is developed as an open platform to attract industry and academia to collaborate on the benchmark and further development of RTFE. The project site is available at *https://github.com/decis-bench/febench*.

1 INTRODUCTION

Online AI-driven applications are emerging and are expected to dominate the AI market in the near future (e.g., accounting for 44% of the market share of AI products by 2030) [16]. As one of the most important components of such applications, real-time feature extraction (RTFE) aims to timely compute features over the incoming new data tuples, which plays an important role in producing high-quality prediction, a.k.a. *"fuel for AI systems"* [21, 28, 29, 42]. However, as advanced machine learning such as deep learning is emerging in many applications, RTFE needs to compute a large number of real-time features (e.g., 662 features for fraud detection [20, 43], 114 features for online recommendation [17, 19, 26], and 458 features for sales forecast [33]) and generally takes over 80% execution time of the whole online AI inference service. In the following, we illustrate a few typical example RTFE applications.

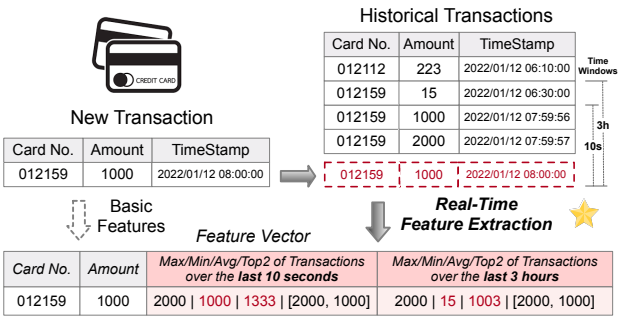


Figure 1: An example of real-time feature extraction.

Example 1 (Fraud Detection). *For banking scenarios, it is vital to identify fraud behaviors as soon as actions like multi-location withdrawals occur, otherwise it may cause serious financial loss (e.g., wasting $2.2 billion one year reported by IRS [10]). 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 3 hours, it may increase the possibility that a fraud action just has occurred. Apart from the average values, it has computed 8 aggregation features (Top 2 is a customized operator) over the two windows to make the feature vector more informative.*

Example 2 (Online Recommendation). *At websites like NetFlix, personalized recommendations are instantly delivered to the users, such as the videos arranged in the homepage, and the recommended videos after the user has finished one. To ensure great user experience, for each recommendation, it requires to immediately update the features based on both a large number of data sources (e.g., historical watches, video information, similar users) and current user actions (e.g., new watches and comments).*

*Both authors contributed equally to the paper.

SIGMOD '23, June 12–17, 2023, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526155

```
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)[1].

Example 3 (Sales Forecast). *In retail companies like Walmart, it is vital to predict the product sales and recommend them to proper customers. Under this scenario, RTFE needs to compute both the short-term (e.g., last one hour) and long-term (e.g., last three months) features of user activities so as to reveal the purchasing habits and guide the retailer to prepare products. Moreover, online sales (e-commerce) require to analyze the preference of different users based on the search (unavailable in offline sales) and purchase records (e.g., most clicked products in the past 5 minutes, products with the most coupons right now), which is even more challenging to compute for high-concurrency user requests in ultra-low latency.*

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

From above examples, we can see real-time RTFE is an non-trivial and challenging issue, for which we need to store a large amount of incoming data, perform complex operators over multiple varied-length windows, and support high-concurrency query requests. We have witnessed similar applications in the business cases from 4Paradigm as well as other industry partners (e.g., Huawei, 37GAMES, Akulaku, JD.com). Thus, relevant projects (e.g., Flink [22], Feathr [5], FeatHub [4]) have devoted their efforts to building practical real-time feature extraction systems. We summarize those efforts in the following two kinds of system designs:

(1) General-purpose stream systems (e.g., Flink and Storm): Many companies have tried to directly build their feature extraction systems on general-purpose systems like Flink (e.g., Cloudian, Airwallex, findify AB). Because these systems own both batch and stream processing capabilities, which suit the requirements of the online AI inference services.

(2) Specialized systems for feature extraction (e.g., OpenMLDB and Tecton): There are two kinds of industrial-strength systems [30, 38, 40]. First, some feature extraction systems aim to serve online features by syncing features pre-computed at offline stage, and they are unable to produce real-time features in low latency. Second, some other systems aim to *update the real-time features in online stage*. For example, with features like "whether the user's credit card is locked", the fraud detection model cannot count on offline batch processing with day-level latency to update those features.

A common feature of those system designs is the RTFE procedures can be expressed in SQL like languages. With this declarative interface, data scientists do not specify the physical execution plans and can focus on describing the feature requirements. Figure 2 demonstrates a simplified RTFE query for fraud detection. Overall, it includes three subqueries. The first subquery involves a single-table window operator, which extracts the basic profiling information (e.g., the user credit, highest monthly balances). The second subquery conducts two-table join, which extracts information from multiple base tables (e.g., transaction amounts from the "transaction" table). The third subquery conducts multi-table window operators, which compute new features from the concatenation of two time windows of the "POS_CASH_balance" table. Finally, all the 662 (in total) features from the three subqueries are combined into one feature vector via the strict one-to-one join operators.

Although real-time feature extraction is an emerging trend and widely taken as a necessity to implement the AI models into production, there is not any study about the workload characteristics and benchmarks for RTFE, specifically in comparison with existing database workloads and benchmarks (such as TPC-C). There are three main challenges. First, there are massive valuable open-sourced realistic datasets in the Internet. 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 characters. Second, it is an non-trivial task to design the benchmark to satisfy the four criterion proposed by Jim Gray, which may be contradictory to each other (e.g., adopting more queries may increase the effectiveness, but negatively affect the benchmark simplicity instead). Third, how to implement the benchmark to relevant systems and gain insights of the performance and system designs.

In this paper, we propose a RTFE benchmark based on the practice of 4Paradigm in providing AI solutions for customers from various sectors (including 75 companies in the Fortune Global 500).

How are the RTFE workloads different from existing database benchmarks? We start with an in-depth analysis of the significant differences between RTFE workloads (e.g., over 20 real applications in 4Paradigm as well as the public datasets in popular ML repositories) and the existing database benchmarks (e.g., transactional [3, 13], analytical [15, 34], and hybrid [14, 25] benchmarks) in consideration of the data distribution, task types, and query structures and complexity.

How to design an effective and efficient benchmark? Based on the massive available RTFE workloads, we have cooperated with our industry partners and built a real-time feature extraction benchmark named FEBench (consisting of the selected datasets, query templates, and online request simulator), which meets the four important criteria for a domain specific benchmark proposed by Jim Gray (Section 3.1).

How do we use FEBench to compare different existing solutions? We provide a testbed with many reusable components (e.g., data loader, workload simulator, performance monitor), which can facilitate researchers to design and test systems with lower overhead on design, evaluation and implementation. We utilize 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*) Different choices of implementation techniques cause great performance difference: *OpenMLDB* (specialized system) executes in assembler code and is much faster than *Flink* (general-purpose system) that runs in JVM; (*ii*) The long tail issue is more severe in *OpenMLDB*, which cannnot efficiently write logs and performs bad in extreme cases (e.g., 99th percentile); (*iii*) The number of parallel threads has a greater impact on *OpenMLDB* than *Flink*. And the results show more work needs to be done for future feature extraction systems. *We aim to develop* FEBench *as an open platform to attract industry and academia to collaborate on the benchmark and further development of RTFE, and 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; and we explain how to generate the 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. Finally, we survey the *existing database benchmarks* and explain why we cannot use these benchmarks to evaluate systems for Feature Extraction in Section 2.4.

2.1 Feature Extraction Operators

In real production scenarios, the AI models heavily rely on a large number of cross-joined or non-linear features [35] so as to improve the AI 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 link the tuples of multiple data streams that share the common columns. Note, to avoid too large intermediate joined table that may slow down the online execution, there are operators like *last joins* where the tuples in the left stream can be matched with the latest matched tuple (ordered by the timestamp columns) in the right stream.

Example. In Figure 2, the "information" table is joined with the "transaction" table to obtain features like the historical transaction amounts of the user who has finished the latest transaction, where sharp increase in the amount values may indicate a fraud action.

```
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). In feature extraction, table window is the most common operator (e.g., computing the traffic state over past 5 minutes for ride forecast). It splits any data stream into "buckets" (each bucket can be split by different columns) of finite sizes and ranks the tuples within each bucket. After that, various aggregations can be conducted over these buckets. However, different from traditional stream operators, RTFE usually concatenates the computed features from multiple parts of *the same table (with different window sizes)*, which help to offer features in different time spans.

Example. In Figure 2, the *average amount* features come 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, which can be written 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, table windows from different data sources may share common columns (e.g., 8 common columns in the "information" and "POS_CASH_balance" tables). For the coming data tuple, we can match the tuple with multiple tables, compute the time window in each table, and concatenate different feature results of the table 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 involves more profiling information (e.g., credit-card balance, instalment amount). Thus, when a new tuple is inserted into "information", we can match the two windowed tables with the new tuple and utilize the output features (on the matched tuples) to enrich online inference information. Beyond OpenMLDB, we can realize similar functions in other systems (e.g., 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. There are hundreds of (basic or customized) aggregation functions supported by relevant systems [2, 8, 12]. Apart from the basic functions (e.g., min, max, average), the customized functions for feature extraction (e.g., *top_n_frequency*, *distinct_count*) are useful to generate new non-linear features from the columns within a table window. Based on the practice experience, we showcase five major categories of aggregation functions (popular features) as follows:

- ***Transformation Features:*** For attribute columns in data sources, sometimes we need to convert them into the required formats, e.g., *dayofweek(*)* for obtaining the day of the month in a timestamp, *degrees(*)* for converting radians to degrees.
- ***Accumulated Features:*** To obtain the *accumulated statistics during a period of time*, we can conduct basic *window+count* operators, e.g., the total purchase frequencies of products over the last month.
- ***Preference Features:*** To obtain *whether some activities exist during a period of time*, we can conduct operators like *window+count_ratio* to check the existence and occurrence frequencies of specific items, e.g., the most frequently purchased products over the last month.
- ***Recent Activity Features:*** Recent activity features aim to reveal the changes or distinct values within recent time period compared with past periods. First, they can compute the difference of features in recent tuples with a bit early tuples (e.g., last time cycle). Second, they can compute the distinct values within recent tuples (e.g., max/min/sum values of different item families). Both of the two classes of features could help to reveal *recent state changes*.
- ***Trend Features:*** To reflect the trends in the near future, we can conduct operators like *window+standard_deviation* to compute the occurring distributions of relevant items, e.g., the average sales in last week. These features reveal *the periodic changes* and generally involve longer time spans than the recent activity features.

(5) Constraints. To avoid too many tuples in the windows slow down online RTFE, there are constraints like "maxsize" that limit the maximal tuple number within the corresponding window.

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. Thus, we proposed to utilize the industry-grade automated machine learning (AutoML) techniques, that is to use feature extraction tool to express RTFE in SQL for ease of building AI models [35]. This tool has been implemented in 4Paradigm's commercial product [2] and severed in many real-world scenarios (such as Industrial and Commercial Bank of China, China UnionPay). Moreover, for those generated queries, the data scientists from 4Paradigm have also confirmed the effectiveness. As shown in Figure 3, given an AI application and the source data, it takes four steps to select features and generate corresponding SQL query.

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

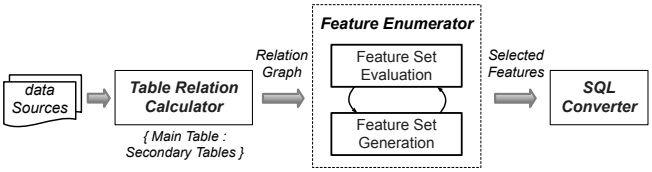


Figure 3: The workflow of RTFE query generation

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

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

Step 1 (initialization): In the table schema, we first identify the main table (storing the stream data) and secondary tables (e.g., snapshots or static 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., the tuple of user profiling information), we directly join the secondary table with the coming tuples in the main table and fetch the whole or part of the join results; and if they are one-to-many relationship (e.g., the historical transactions of a user), we first join the secondary and main tables and than conduct aggregation on the join results, where the adopted aggregation functions depend on the secondary table type (e.g., *count* operator if the secondary table is static; *groupby_count* operator if the secondary table is appendable). Each mapped operator pattern corresponds to a candidate feature (the output column).

Step 3 (feature enumerator): After extracting all the candidate features, we iteratively select useful feature sets by two steps: (*i*) *Feature set generation:* We utilize beam search to explore an extensive search space of the candidate feature sets; (*ii*) *Feature set evaluation:* We evaluate the candidate feature sets based on the task target (e.g., AUC metric for a classification task) and the best is selected as a new solution. This iterative procedure is terminated once some conditions (e.g., maximum iteration time) are met.

Step 4 (SQL converter): Finally, we convert the selected features into an semantic-equivalent SQL query. When serving the application, we implement the SQL query into the feature extraction system (e.g., "DEPLOY" command in OpenMLDB) and rely on the system to efficiently execute the query. We list the generated SQL queries for real datasets at https://github.com/decis-bench/febench.

2.3 Systems for Feature Extraction

A large number of products have been developed to support feature extraction (e.g., *Michelangelo* in Uber [38], *Airbnb* in Zipline [23], *Feathr* in Microsoft [5], *OpenMLDB* in 4Paradigm [8]). Here we

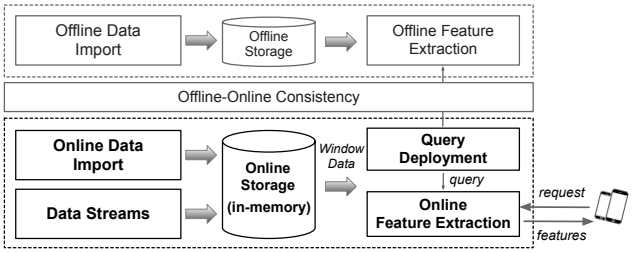


Figure 4: The general architecture of RTFE systems.

clarify the definition of "real-time feature extraction" as *on-demand feature computation and request response in the online stage*. And so systems that pre-compute the feature values in offline and store the recent features in caching for online requests are not in the scope of this paper. Generally, the overall architecture for real-time feature extraction is composed of three main modules (Figure 4).

(1) Online feature extraction. Online AI applications require real-time processing of new data streams into feature values so models can make timely predictions. Thus, the online storage is memory-based, which stores the feature values for low-latency lookup during inference. They typically only store the latest feature values, essentially modeling the current state of the world. Online stores support multiple copies for the same table data to ensure high availability. Besides, with the RTFE query deployed ahead of time (Section 2.2), they utilize the optimized query engine to process online requests. Different from traditional stream systems, they enhance the procedure from multiple aspects, e.g., supporting (i) double-layered skiplist to efficiently compute TopN operators over time windows and (ii) overlapped window reuse to enhance the data requests [21, 24] (see the example execution plan in Figure 13).

(2) Offline feature extraction is typically used to persist months or years of feature data and conduct batch model training. Offline feature data is often stored in data warehouses or data lakes (e.g., Hadoop, BigQuery, Snowflake, Redshift). And they share the same feature extraction query with the online part.

(3) Offline-and-online consistency. The ML 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. Thus, the computing logic of offline and online feature should be consistent. For example, engineers from Varo (an online bank from the US) have found that inconsistent execution definitions of "account balance" at offline and online stages cause significant model quality degradation at production [7]. In their project, the account balance of yesterday is used at offline, while the current account balance is used for online. Therefore, a consistent view of feature definition across the offline and online feature extraction is a must for an industry-grade feature extraction system.

2.4 Existing Benchmarks

System benchmarking has been an active area of both the research and industry communities [18, 41]. And most of the standard benchmarks are derived from the real data and typical queries, including transactional benchmarks [3, 13], analytical benchmarks [15, 34], and hybrid transactional/analytical benchmarks [14, 25, 31]. Next we introduce these benchmarks and explain the challenges in evaluating real-time feature extraction (Table 3).

2.4.1 Transactional Benchmarks. Online transaction processing benchmarks aim to evaluate the capability of maintaining the business data and processing a large number of small-sized transactions.

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

Example. TPC-C [13] simulates a real transactional scenario, where a company (with multiple warehouses and sales districts) processes the orders issued by the clients. TPC-C provides a write-heavy workload, which contains 92% write operators over 9 tables. TPC-C tables can be scaled to different sizes, which are 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 very simple*, e.g., single table access and does not contain the joins or unions of multiple tables.

2.4.2 Analytical Benchmarks. Online analytical benchmarks aim to evaluate the performance of complex data analysis tasks.

Scenarios. Different from OLTP systems, OLAP systems aim to efficiently process large-scale table scans and aggregations; join data from multiple tables; and perform multi-dimensional operators (e.g., with at most 3-level subqueries).

Example. TPC-H [15] simulates a real scenario, where a wholesale supplier performs deliveries all around the world. The workload contains 22 business queries, each of which perform complex data operators (e.g., joins, subqueries). TPC-H does not consider write operators and the dataset size does not change during workload execution. The benchmark metric is both throughput (e.g., QphH) and total execution latency. Besides, TPC-DS is an OLAP benchmark that simulates a more complex real scenario than TPC-H. In TPC-DS, it contains 99 queries, which involve structures like table unions that do not exist in TPC-H queries. However, these OLAP benchmarks only test the batch processing capacity and does not consider online processing, i.e., (i) the tables are loaded in batch and do not have incoming data by timestamp; (ii) the queries involve complex operator patterns over global data and generally take seconds to minutes to finish. Thus, it requires *new benchmark to support complex analytics over data streams.*

2.4.3 HTAP Benchmarks. Hybrid transactional/analytical processing benchmarks aim to efficiently support both operational (small transactions with high update ratio) and analytical workloads (complex queries) within the same system.

Scenarios. First, HTAP systems need to perform real-time data analytics between online transactions. Second, they need to prevent the interference of the analytical queries over the dynamically-changing data tables.

Example. CH-benCHmark [25] provides a mixed workload based on both TPC-C and TPC-H benchmarks. It enables two types of clients to separately serve transactional and analytical queries. To allow analytical queries to access transactional tables, it merges

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

	Workload	Response Time	Time-Series	#Aggregations	#Windows	#Subqueries	#Joins
TPC-C [13]	transaction	milliseconds	×	0 ~ 2	×	×	×
Sysbench [3]	transaction	milliseconds	×	×	×	×	×
TPC-H [15]	analytics	seconds; minutes	×	0 ~ 8	×	0 ~ 4	0 ~ 8
JOB [34]	analytics	minutes	×	1 ~ 6	×	×	0 ~ 17
CH-benCHmark [25]	hybrid	seconds; minutes	×	0 ~ 8	×	0 ~ 4	0 ~ 8
StreamBench [46]	streaming	subseconds	✓	0 ~ 2	~ 1	×	0 ~ 2
FEBench	RTFE	milliseconds	✓	5 ~ 800	0 ~ 10	1 ~ 18	0 ~ 14

the two schemas into a single one. For real-time feature extraction scenarios, HTAP benchmarks meet the same challenges as OLAP benchmarks, i.e., lacking support of the time-series data and taking seconds to minutes to execute the analytical queries.

In summary, we can see the workloads of existing benchmarks are completely 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, with which we can simulate the real coming stream data (Section 4); (*ii*) how to identify the critical differences of RTFE and existing database workloads (Section 5); (*iii*) how to select RTFE workloads for a domain-specific benchmark with representative query structures and complexity (Section 6).

3 BENCHMARK OVERVIEW

3.1 Design Goals

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

Relevance. The benchmark covers a wide range of feature extraction behaviors as well as different operator complexities. We collect over 100 real feature extraction workloads from different sources, i.e., *45 Kaggle datasets [11], 11 Tianchi datasets [9], 8 KiltHub datasets [6], 28 UCI ML datasets [1], and 26 applications in 4Paradigm*, which cover the major feature extraction operators (Section 2.1) and scenarios that highly rely on real-time feature extraction, e.g., ride prediction, healthcare, energy consumption, sales forecast, and fraud detection.

Simplicity. The benchmark is understandable and has eliminated redundant tests. Out of the collected RTFE datasets, we cluster the queries and only take a small number as the query templates, which reveal the typical RTFE operator patterns. Besides, it reduces the redundancy of testing on similar datasets. For each query template, we describe the semantics and operator patterns, and try to make them easy to understand (Section 5).

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

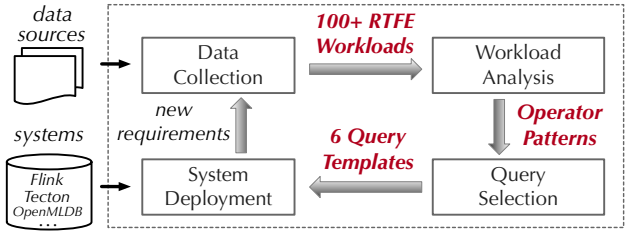


Figure 5: The workflow of FEBench generation.

Scalability. The benchmark includes datasets of various sizes. And the datasets can be scaled to different data sizes by following the real distributions of incoming data (Figure 6).

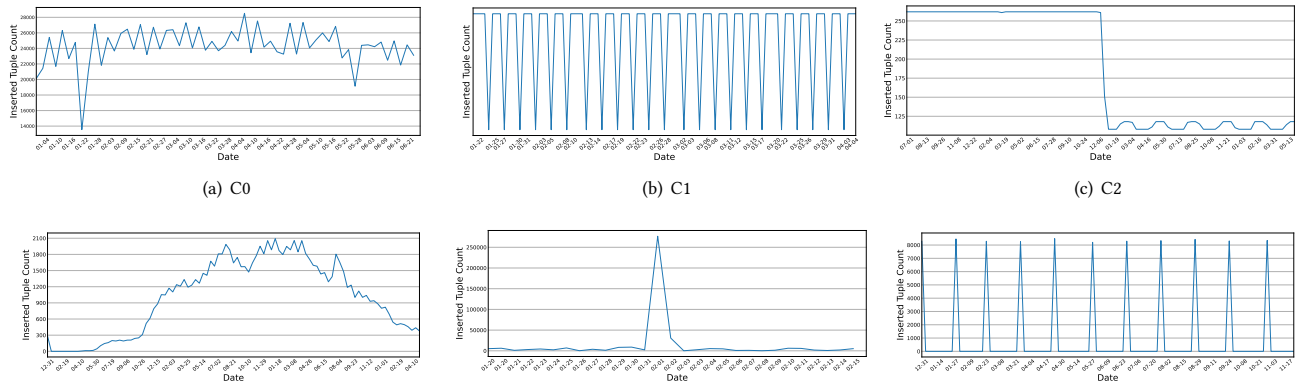
3.2 Benchmark Methodology Overview

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

Workload Collection. This module includes two parts. First, to cover as many AI applications as possible, with the search engines, we try our best to find all the public AI repositories (e.g., Kaggle, Tianchi, UCI ML, KiltHub). For each repository, we extract datasets that satisfy the settings of real-time feature extraction, i.e., (*i*) tabular data; (*ii*) with at least one timestamp column; (*iii*) the data size is large enough to support minutes of tests. Besides, we obtain 26 real scenarios in 4Paradigm [24]. For each collected dataset, we either utilize the manual-crafted feature extraction query by data scientists, or adopt the automatic query generation tool (Section 2.2) whose output queries can (*i*) reveal the important feature information and are manually verified and (*ii*) achieve higher execution efficiency than that written by data scientists.

Workload Analysis. With the collected datasets and queries, we compare with the existing database benchmarks, 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), and summarize the major data and query characters and task targets in feature extraction.

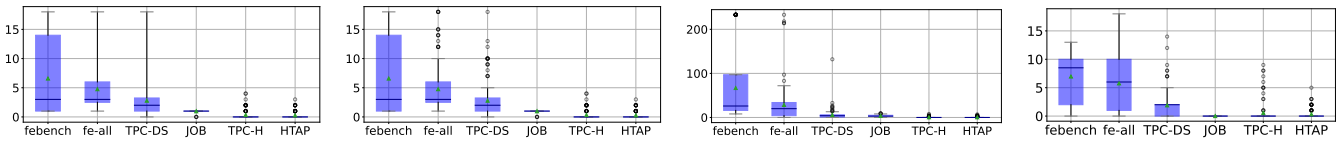
Workload Selection. Some collected datasets may have similar testing effects. To ensure both the simplicity and effectiveness of



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

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

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



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

Figure 7: Contrast of the query operators. Note that FEBench denotes the selected 6 RTFE query templates, FE-all denotes the whole set of 118 RTFE queries, TPC-DS/JOB/TPC-H are analytical queries.

the benchmark, we need to cluster all the RTFE queries into templates so as to combine queries with similar operator patterns and scenario requirements. First, we rank the importance of different operators to the computation performance with logistic regression, where each operator is assigned with an importance weight. Second, with the weighted operators, we take them as the query feature vector and utilize DBSCAN [32] to cluster the origin queries into a number clusters, each of which denotes 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 finally gain 6 clusters with relatively few outliers. Besides, around the centroid of each cluster, we try to pick datasets that come from different scenarios to better cover the feature extraction cases. In FEBench, there are six workloads coming from five typical AI applications and are of various operator patterns.

Deployment on Target Systems. With the selected workloads (i.e., query templates and real datasets), we implement them on suitable systems, i.e., general-purpose systems like Flink [22], and specialized systems like OpenMLDB [24]. Under the same benchmarking environment, we test the performance of these systems with the selected query templates and find some interesting points (e.g., the balance between execution efficiency and system compatibility) by profiling the execution results.

Workflow. As shown in Figure 5, first we extract the suitable workloads from both public data sources (e.g., popular ML communities and contests) and practices, which involve numerous different machine learning datasets, and take great manual efforts to filter unsuitable datasets. Second, with the collected workloads, we compare them with the workloads of other benchmarks, and provide

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}

observations of the unique characters of the RTFE workloads. Third, since too many datasets will increase the evaluation overhead, we cluster the origin queries and select 6 most representative query templates. Finally, we describe how to implement the benchmark to the feature extraction systems and test their performance from different aspects.

4 WORKLOAD COLLECTION

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

4.1 Dataset Collection

> Finding 1. *The selected 118 datasets cover 5 common feature extraction scenarios, which are of relational data and own various timestamp distribution (e.g., cycles, sudden bursts). In the datasets, the numbers of tables are within [1,10], and the data sizes range from 2MB to 10TB (Table 4).*

We spent over 2 person years to comprehensively analyze over 1000 AI applications from popular open data sources (e.g., Kaggle [11], Tianchi [9], UCI ML [1], KiltHub [6]). Taking *Kaggle* as an example, which is one of the biggest online community of data scientists and ML practitioners. There are 500+ competitions and corresponding real decision-making tasks and datasets (last check at September 20, 2022). For real-time feature extraction, any selected dataset must satisfy the following requirements:

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

(2) Timestamp feature: Real-time feature engineering needs to update the computed features based on most recent incoming data. And so any selected dataset must contain the "timestamp" feature, which simulates the different coming data distributions in real online scenarios (e.g., burst incoming data in Figure 6 (e), cycles in Figure 6 (b));

(3) Data scales: We require at least one table with timestamps in the dataset (generally the main table) contains over 1×10^6 tuples, which are used to simulate the real data incoming scenarios and test the performance for minutes.

Based on the requirements, we select from all the open-sourced datasets in different sources. For example, from the 500+ Kaggle competitions, we first remove datasets not in tabular format. For each tabular-data dataset, we look up the "timestamp" column and check the data distributions (e.g., the average interval between two tuples is no longer than 1 minute) one by one, and finally we extract 45 potentially useful datasets that meet the requirements. Overall, 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.*

As shown in Table 4, these collected datasets cover a large and typical range of data distributions. First, the number of tables ranges from 1 to 10, and the dataset sizes range from 1MB to 10TB. Second, most datasets contain single tables (the "training" table), whose queries do not involve the direct joins of multiple tables but may need to combine multiple table windows of different sizes. Many datasets also contain 2-6 tables, whose mapped relations (Section 2.2) are relatively complex and the queries may involve tricky subqueries of dozens of levels. Third, the sizes of most datasets are no larger than 10GB, where only the tuples within table windows are required and these tuples are sufficient to support the incoming data simulation for minutes.

5 WORKLOAD ANALYSIS

With the collected workloads, next we demonstrate our analysis of the unique workload characters 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 characters 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 (#-keyword, #-aggregates) observed was (477, 38) for TPC-H, (883, 73) for TPC-DS 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 [25] have similar characters 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 explain how to generate the FEBench benchmark by selecting templates out of the collected 118 RTFE queries, which contain redundant operators and structures and may affect the benchmark portability; and provide scenario analysis of FEBench.

6.1 Query Template Selection

With the 118 collected datasets, we first generate the feature extraction queries for the collected datasets (see Section 2.2). Next, we select representative queries based on both the clustering results and scenario characters (Figure 8).

Specifically, we utilize the clustering algorithm *DBSCAN* to divide the 118 generated RTFE queries based on their feature vectors. First, the *DBSCAN* algorithm does not need to manually set the cluster number and can better reflect the query distribution. Second, for

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

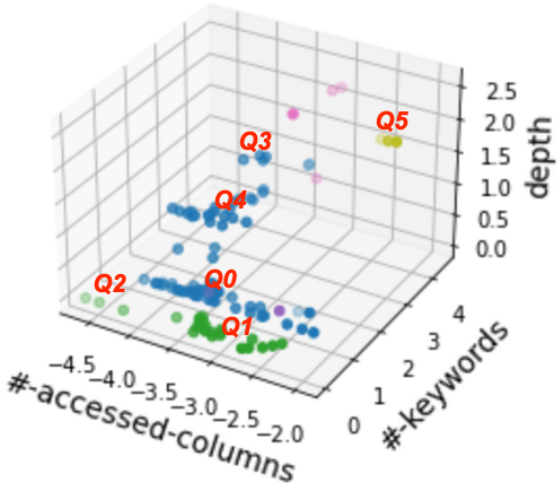


Figure 8: Query Clustering Analysis. Divide 118 RTFE queries into 6 clusters (templates) with the DBSCAN algorithm. Each dimension is normalized into a relatively dense range.

each RTFE query, the feature vector include (i) the number of output columns, which reflects the result scales and (i) the total number of query operators, which reflects overall query complexity and (iv) the occurrence frequencies of complex operators (e.g., joins, windows, set unions), which reflect the detailed operator-level complexity. Note different from traditional database queries, RTFE queries involve table windows and so the query complexity is not directly affected by the scale of batch-loaded data.

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 the 6 main RTFE application tasks and have unique operator patterns, which are closely relevant to the data characters.*

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 characters 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 [22].

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 complete 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 weathers.

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 these two platforms: OpenMLDB [24] and Flink [22].

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-9999). 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

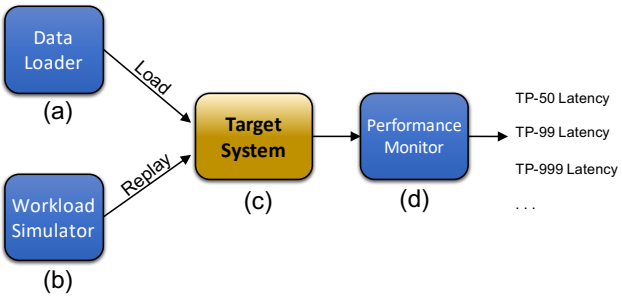


Figure 9: The overview of FEBench pipeline.

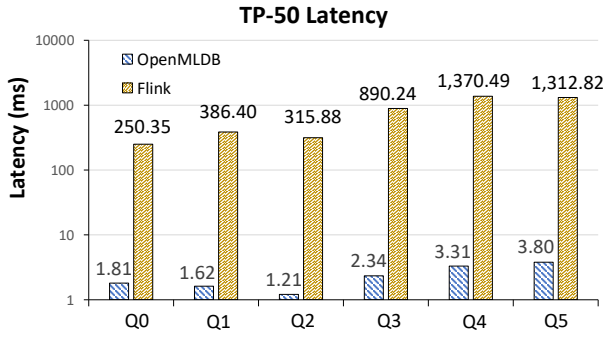


Figure 10: The TP-50 latency of all tasks.

Flink is 1.15.2, and we deploy one job manager node and three task 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

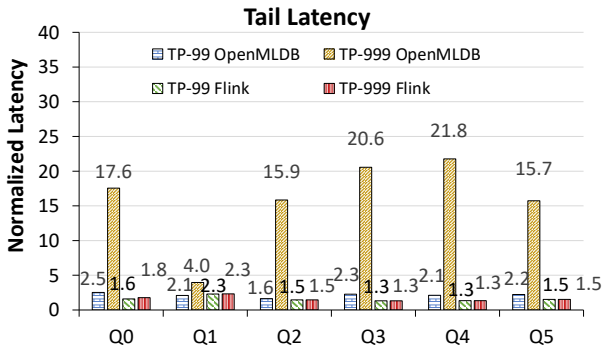


Figure 11: Normalized tail latency (All values normalized to that of the TP-50 latency).

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 [24]. 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 [24]. 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 12 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 12, OpenMLDB's TP-50 latency does not raise rapidly when the threads number

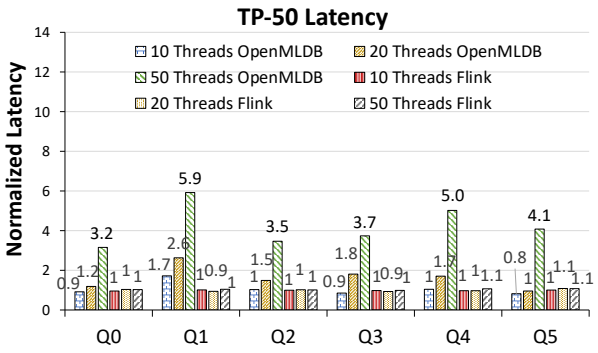


Figure 12: Normalized TP-50 latency (All values normalized to that of the 5 threads).

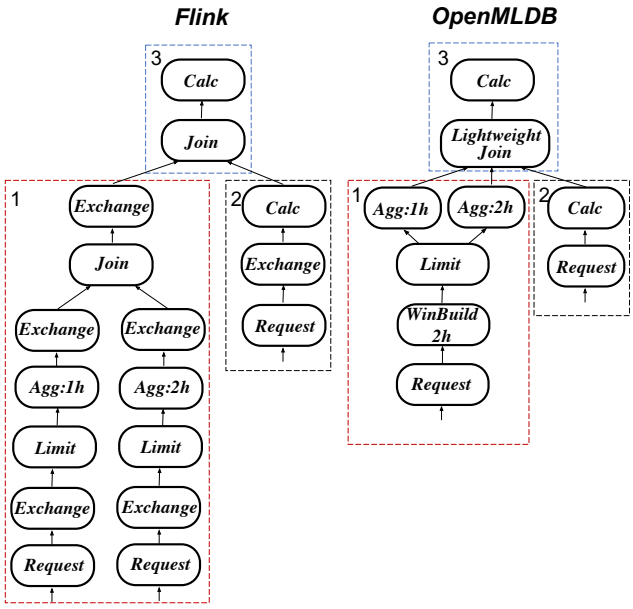


Figure 13: Example Execution Plans (Q0).

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 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 13, 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 13, 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 13). 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 can significantly reduce the overhead of future feature extraction system designs.

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://flink.apache.org/.
[3] https://github.com/akopytov/sysbench.
[4] https://github.com/alibaba/feathub.
[5] https://github.com/feathr-ai/feathr.
[6] https://kilthub.cmu.edu/.
[7] https://medium.com/engineering-varo/feature-store-challenges-and-considerations-d1d59c070634.
[8] https://openmldb.ai/.
[9] https://tianchi.aliyun.com/.
[10] https://www.irs.gov/pub/irs-prior/p3415–2021.pdf.
[11] https://www.kaggle.com/competitions.
[12] https://www.tecton.ai/.
[13] https://www.tpc.org.
[14] https://www.tpc.org/tpcds/.
[15] https://www.tpc.org/tpch/.
[16] Forecast: The business value of artificial intelligence. In *Gartner*, 2018.
[17] S. P. Anderson. Advertising on the internet. *The Oxford handbook of the digital economy*, pages 355–396, 2012.
[18] 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.
[19] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-based systems*, 46:109–132, 2013.
[20] R. J. Bolton and D. J. Hand. Statistical fraud detection: A review. *Statistical science*, 17(3):235–255, 2002.
[21] J. Cai, J. Luo, S. Wang, and S. Yang. Feature selection in machine learning: A new perspective. *Neurocomputing*, 300:70–79, 2018.
[22] 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.
[23] S. Charrington. Machine learning platforms.
[24] 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.
[25] 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.
[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.