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

Xuanhe Zhou*
Tsinghua University
zhouxuan19@thu.edu.cn

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

Kunyi Li
Tsinghua University
lkg19@thu.edu.cn

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

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

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

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

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

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

Yuqiang Chen
4Paradigm Inc.
chenyuqiang@4paradigm.com

ABSTRACT

As the use of online AI inference services rapidly expands in various applications (e.g., fraud detection in banking, product recommendation in e-commerce), real-time feature extraction (RTFE) systems have been developed to compute the requested features from incoming data tuples in ultra-low latency. Similar to relational databases, 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 (e.g., concurrent transactions in TPC-C). In this paper, we study the RTFE workload characteristics using over one hundred real datasets from open repositories (e.g. Kaggle, Tianchi, UCI ML, KiltHub) and those from 4Paradigm and its customers. The study highlights the significant differences between RTFE workloads and existing database benchmarks in terms of application scenarios, operator distributions and query structures. Based on these findings, we propose to develop a real-time feature extraction benchmark named FEBench based on the four important criteria for a domain-specific benchmark proposed by Jim Gray. FEBench consists of selected representative datasets, query templates, and an online request simulator. We use FEBench to evaluate the effectiveness of feature extraction systems including OpenMLDB and Flink and find that each system exhibits distinct advantages and limitations in terms of overall latency, tail latency, and concurrency performance.

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 [15]). As a crucial component of AI 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"* [21, 31, 32, 46, 53]. However, with the rise of advanced machine learning techniques (e.g., deep learning) and the increasing complexity of AI-driven businesses, the number of features that must be computed in real-time has significantly increased (e.g., over 600 features for fraud detection [20, 47], over 100 features for online recommendation [17, 19, 28], over 400 features for sales prediction [36]). RTFE often accounts for a huge proportion of execution time of the online machine learning pipeline (e.g., taking 70% time in the sales prediction service of an online car purchase platform according to the practical experience in 4Paradigm[1]). To provide a better understanding, we present a few examples of typical RTFE applications.

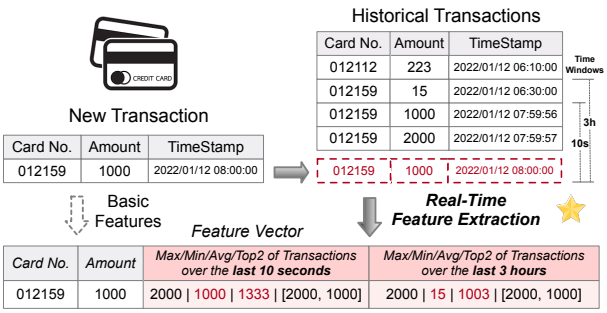


Figure 1: An example of real-time feature extraction.

Example 1 (Fraud Detection). *For the banking industry, it is crucial to identify fraudulent activities (such as multi-location withdrawals) in real time, so as 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 time 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 Advertising). *In companies like Criteo, it is essential to dynamically optimize the placement of advertisers' contents for each internet user (e.g., conducting 950 billion daily optimizations, each of which taking place within 50ms). To reach this goal, it is challenging to rapidly update the features (for tasks like clicking probability forecasting of each advertisement) based on a large number of data sources (e.g., historical browsing records, users*

*Both authors contributed equally to the paper.

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

with similar browsing records) and current user actions (e.g., most recent browsing records). The real-time feature update process helps to deliver fast responses to assist advertisers reach more users.

Example 3 (Sales Prediction). *In retail companies like Walmart, accurate prediction of product sales and 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.*

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 (e.g., for aggregation features) 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 both the commercial customers and open-source community partners of 4Paradigm (e.g., Intel, 37GAMES, Akulaku, and JD.com)[2].

These challenges and applications have prompted the development of real-time feature extraction systems or components in various projects (e.g., Flink [22], Feathr [4], FeatHub [3], and OpenMLDB [7]). These efforts can be broadly categorized into two types of system designs. *(1) General-purpose stream processing systems (e.g., Flink):* Many companies have attempted to construct their feature extraction systems on top of general-purpose systems like Flink. These systems possess both batch and stream processing capabilities, making them suitable 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 [33, 41, 44]. 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 long latency.

Similar to relational databases, 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. A natural question is: are RTFE workloads different to existing database workloads? On the other hand, although RTFE is increasingly viewed as essential for deploying AI models in production, there is currently no research on the workload characteristics and benchmarks for RTFE, and especially the comparison with existing database workloads and benchmarks.

In order to answer the above question, this presents three main challenges. ❶ A benchmark should be rooted from real RTFE workloads. Due to the massive number of valuable real datasets available online, 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 (**C1**). ❷ It is a non-trivial task to design the benchmark to satisfy the four criteria proposed by Jim Gray [29], which can be contradictory (e.g., adopting more queries may enhance the effectiveness, but negatively affect the benchmark simplicity) (**C2**). ❸ It is crucial to deploy the benchmark in both general-purpose and specialized systems and gain insights into the system designs with the benchmark (**C3**).

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 collected RTFE workloads (i.e., over 100 suitable datasets extracted from over 1000 public machine learning tasks) and existing database benchmarks (e.g., transactional [2, 12], analytical [14, 37], and hybrid [13, 25] benchmarks) in consideration of the data distribution, task types, and query operators and structures (§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 consists 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 3 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 develop 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).

The initial efforts in this paper are promising to grow the research and development interests to RTFE. It is our long-term goal 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, and then discuss why existing database benchmarks cannot be used to evaluate feature extraction systems.

[2] github.com/4paradigm/OpenMLDB/discussions/707

```
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' between 100s
     preceding and 0s),
   balance_0_10 AS (
     UNION 'POS_CASH_balance'
     PARTITION BY 'ID_CURR'
     ORDER BY 'ingestionTime' between 10s
     preceding and 0s) ) AS out2
ON out0.reqId_1 = out2.reqId_4;
```

Figure 2: Feature Extraction Query (for Fraud Detection)

2.1 Feature Extraction Operators

As mentioned in Introduction, the RTFE procedures can be expressed using SQL like languages. As shown in Figure 2, a simplified RTFE query for *Example 1 (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 customized join operator to efficiently extract information from one or multiple tables ordered by timestamps, like the transaction amounts from the transaction table. The third subquery uses multi-table window operators to calculate temporal features from two time windows (10s and 100s) of the POS_CASH_balance table.

Note the example queries in this paper follow the SQL standards of OpenMLDB. Other systems like Flink have similar SQL grammars.

To achieve the aforementioned procedures, various RTFE operators corresponding to distinct real-time features are available. Here we showcase five main categories of operator patterns.

(1) Table Joins (basic information). Join operators are used to link tuples of multiple data streams that share common columns. Different from database joins, to reduce the need for a large intermediate joined table and the costly tuple sorting associated with it (which can slow down online execution), operators like *last join* match the tuples in the left stream with the latest matched tuple in the right stream (pre-ordered by the timestamp column).

Example. In Figure 2, the "information" table is joined with the "transaction" table to obtain features like the historical transaction amount of a user who just completed the latest transaction, where a sudden increase in the amount 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 (recent activities from single source). 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 (recent activities from multiple sources). Similar to traditional joins, multi-table windows enable time windows from different data tables that share common columns (e.g., 8 common columns in the "information" and "POS_CASH_balance" tables). By matching an incoming data tuple with these tables, we can compute the time windows in each table, and concatenate the output window features so as to enrich the 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 the "information" table, we can match the two tables in time windows (i.e., within last 100 tuples) with the new tuple and leverage the output features (on the matched tuples) to

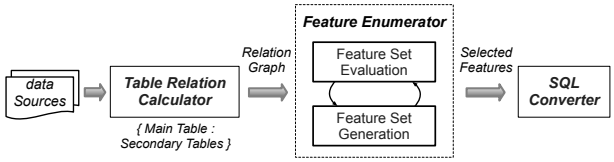


Figure 3: The workflow of RTFE query generation

enrich online inference information. Note the multi-table window operator is not limited to OpenMLDB, as similar capabilities can be realized in other systems like point-in-time joins in Tecton.

```
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[3]:

• ***Transformation Features*** are used to convert attribute columns in data sources into the required formats. This can include operators 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 further 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 Status 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 status features, it is possible to better understand the current state of the data and make informed decisions based on these insights.

[3]Check detailed features in github.com/decis-bench/febench/tree/main/features

Table 1: Relationships (with the main table) and the mapped operators. *Events* indicate recent activities; *Status* denotes long-term properties.

Secondary Table	Relationship	Operator Pattern
static/attribute table	one-to-one	last join
	one-to-many	aggregation + left join
appendable table	one-to-many	aggregation (events)
snapshot table	one-to-one	last join
	one-to-many	aggregation (status)

• ***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 staus 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. Efficient real-time feature extraction requires avoiding an excessive number of tuples within the time windows, as this may slow down the process. For example, the constraint "maxsize" can be used to limit the number of tuples included within the windows. By appropriately setting constraints, it is possible to balance (i) the need for computing effective features and (ii) maintaining the efficiency and responsiveness of the feature extraction process.

2.2 Existing Database Benchmarks

System benchmarking is a highly active area of both research and industry communities [18, 45]. Most standard benchmarks are derived from real data and typical queries, including transactional benchmarks [2, 12], analytical benchmarks [14, 37], and hybrid transactional/analytical benchmarks [13, 25, 34].

2.2.1 Transactional Benchmarks. Online transaction processing (OLTP) benchmarks evaluate the ability to maintain business data and process high-concurrency transactions, which generally involve a limited number of tuples.

Scenarios. The scenarios of OLTP benchmarks involve two critical aspects. 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 [12] 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 [27]. 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. Thus, in TPC-C, the operator patterns are relatively simple (e.g., with single table access and no unions of multiple tables).

2.2.2 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) [16, 38, 54].

Example. TPC-H [14] 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 operators like table unions that do not exist in TPC-H queries. However, these OLAP benchmarks only test batch processing capacity over global data, and also do not consider online evaluation over data streams.

2.2.3 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 [25] 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 table 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 database benchmarks fail to (i) simulate RTFE scenarios (e.g., stream processing for feature extractions) or (ii) support complex operator patterns over time windows or (iii) conduct evaluations in online mode. RTFE systems require *a new benchmark to evaluate real-time complex analytics over data streams.*

3 BENCHMARK OVERVIEW

3.1 Design Goals

We design the feature extraction benchmark by following the *4 benchmark design criteria* proposed by Jim Gray [29].

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 prediction, 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 workloads and only use a small part as query templates, which represent typical RTFE operator patterns and reduce redundant tests on similar workloads. Besides, for each query template, we separately describe the data distributions, query semantics, and operator patterns for ease of understanding.

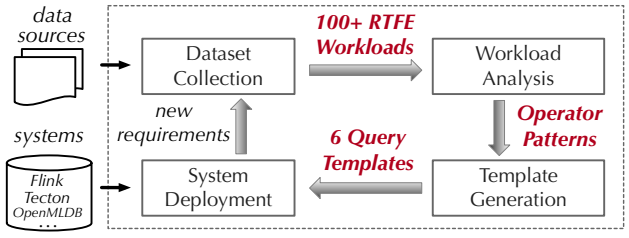


Figure 4: The workflow of FEBench generation.

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 functions with the combinations 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 8).

3.2 Benchmark Methodology Overview

Based on the design goals, we build the benchmark in a workflow of four steps (see Figure 4). Firstly, we extract suitable workloads from a variety of public and industry-grade data sources. Secondly, we compare the collected workloads with other benchmarks, highlighting the unique characteristics of the RTFE workloads. Thirdly, based on the four benchmark criteria from Jim Gray, 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.

Dataset Collection. This module includes two parts. First, we search for datasets in various public AI repositories to cover as many AI applications as possible. We collect datasets that meet the RTFE 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 real datasets from 4Paradigm [24]. For each collected dataset, we synthesize the RTFE query using our automatic query generation tool (Section 3.3).

Workload Analysis. We compare the collected datasets and queries with existing database benchmarks. We analyze the data distributions (e.g., table schema, incoming data patterns) and query structural differences (e.g., the types/numbers/patterns of supported operators). Using this analysis, we summarize the major data/query characteristics in real-time feature extraction.

Template Generation. 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 utilize logistic regression to rank the importance of different query features (e.g., the nested level, the operator number) based on their impact on execution latency, and assign each feature a weight. Then, we apply DBSCAN [35] to divide the origin queries into clusters based on these weighted features. Note that, to balance between the benchmark simplicity and effectiveness, we tune the DBSCAN parameters (e.g., *eps* controls the minimal distance of queries within the same cluster, *min_samples* controls the minimal queries in the same cluster) and try to pick

Table 2: Histogram information of 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

queries that come from different scenarios around the centroid of each cluster to better cover diversified RTFE cases.

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 [22] and specialized systems like OpenMLDB [24]. More details can be found in Section 3.4. Under the same 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 in finer granularity.

In the remainder of this section, we present the details about automatic query generation and target test systems, and leave the details about dataset collection, workload analysis and system deployment in Sections 4, 5, and 6, respectively.

3.3 RTFE Query Generation

Next we introduce how to generate the RTFE queries. In practice, it is laborious and time-consuming for data scientists to sample and try out different feature combinations. To simplify the process, we utilize the industry-grade automated machine learning (AutoML) technique, which can express RTFE in SQLs for ease of building AI models [30, 39]. This tool has been implemented in 4Paradigm's commercial product[4] and served in many real-world scenarios (such as Industrial and Commercial Bank, UnionPay).

In this work, the data scientists at 4Paradigm have verified the validity and effectiveness of generated queries for our collected datasets. Given an AI task and source data, the selection of features and generation of RTFE queries involve four steps (Figure 3):

Step 1 (initialization): We first identify the main table (storing the stream data) and secondary tables (e.g., static/appendable/snapshot tables) in the dataset. Next, we enumerate the one-to-one/one-to-many relations (corresponding to different RTFE operators) of columns within the main table and secondary tables, like columns with similar names or key relations.

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 of one-to-many relationship (e.g., historical transactions of a user), we first join the secondary and main tables and then perform aggregations on the join results. The adopted aggregation functions depend on the secondary table type (e.g., *count* operator for static tables, *groupby_count* operator for appendable tables). Each mapped operator pattern corresponds to a candidate feature (represented by an output column).

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

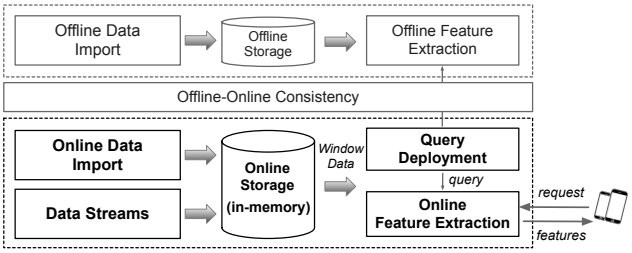


Figure 5: The general architecture of RTFE systems.

Step 3 (feature enumerator): After extracting all the candidate features, we utilize beam search [43] to iteratively generate effective feature sets. That is, we first initialize a root node (level 0) denoting the basic features in origin tables. Next, in each iteration, we select one most promising node N to expand nodes of next level (e.g., adding a candidate feature to the feature set of N) based on metrics like AUC (measuring the model inference accuracy with the updated feature set [26]) and selection frequency (the occurrence numbers of different columns). This iterative procedure is terminated once a specified condition (e.g., the maximum iteration time) is met and the feature set of the best leaf node is chosen.

Step 4 (SQL converter): Finally, the selected features are converted into a semantically equivalent SQL query, which needs to integrate into the feature extraction system before evaluation (e.g., the "DEPLOY" command in OpenMLDB). Note for a new feature extraction system, the SQL converter can be easily adapted provided the SQL grammar of this system is known.

3.4 Target Test Systems

Real-time feature extraction refers to *on-demand RTFE query execution and request response in online stage*. Although there are various products that support feature extraction (e.g., *Michelangelo* in Uber [41], *Zipline* in Airbnb [23], *Feathr* in Microsoft [4], Tecton [11], *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 5, a RTFE system typically has three main modules:

(1) Online feature extraction is essential for real-time processing of incoming stream data into features that enable timely model inferences. The online storage is mostly 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 3.3), the systems use an optimized query engine to process online requests. Different from traditional stream systems, the query engine enhances the procedure through various optimization designs, such as supporting (i) data structures like double-layered skiplist to optimize window operators by sorting tuples based on both the key column and time ranges [24] and (ii) overlapped window reuse that helps to enhance data requests over multiple windows [21] (see the example query 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. Offline feature extraction shares the same feature extraction query as the online module.

(3) Offline-and-online consistency. Machine learning models require 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 inference 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.

Our benchmark is portable to different RTFE systems. As a start, we perform deployment and detailed studies on two systems: Flink [22], a general-purpose system, and OpenMLDB [24], a specialized system.

4 DATASET COLLECTION

In this section, we explain how to collect and prepare the RTFE datasets together with the feature extraction queries.

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

We spent over ***2 person years*** to comprehensively analyze various 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 on 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 column: Real-time feature extraction requires the updating of computed features by the most recent data. Thus, any selected dataset must contain a "timestamp" column that simulates the various incoming data distributions in real online scenarios (e.g., periodic cycles for the flu forecast task in Figure 8 (b), random bursts for the loan payment task in Figure 8 (e));

(3) Data scales: We need the dataset that includes at least one table with timestamps (usually the main table) and contains 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 a total of 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 feature extraction query (see Section 3.3).

These RTFE datasets have three characteristics. First, *the collected datasets cover a wide range of data distributions.* As shown in Table 2, 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 (full data is stored in HDFS for batch training) and (ii) only the data tuples within time windows are required. Note we also have relatively large datasets (e.g., over 10TB), whose RTFE queries do not involve windows but join the incoming tuple with full static tables.

5 WORKLOAD ANALYSIS

In this section, we demonstrate the analysis of the unique workload characteristics of RTFE in comparison with typical database benchmarks.

5.1 Observations

Based on the discussion in Section 2.2, we further analyze the detailed operator distributions of RTFE workloads (FEBench) with transaction (TPC-C), analytic (TPC-DS, TPC-H, JOB), and hybrid workloads. The results are shown in Figure 6.

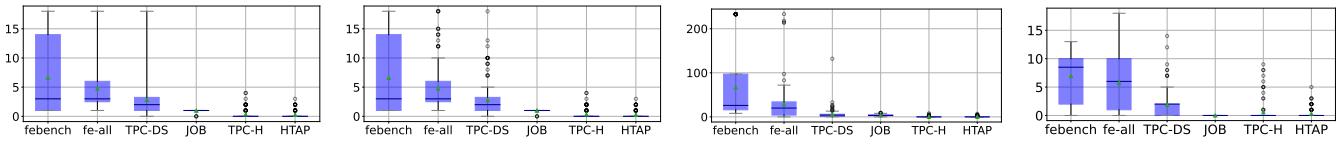
> Finding 2. *Among the 118 queries, they support 15 typical operators and various customized aggregations. Compared with transactional/analytical queries, most of the RTFE queries involve much more complex query structures over time windows.*

RTFE Operators vs Transactional Operators. First, both RTFE and OLTP workloads support high-concurrency queries. However, OLTP supports data update (e.g., 28.0% update queries), 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 RTFE systems focus on the high efficiency of processing complex operators and adopt in-memory architectures.

RTFE Operators vs Analytical Operators. As shown in Figure 6, RTFE queries cover a much larger range of operator space than OLAP queries. To quantify, the highest number of toal operators and aggregations observed in a query is (477, 38) for TPC-H, (883, 73) for TPC-DS, (4969, 233) for RTFE. Besides, most RTFE queries have distinct operator distributions from TPC-H/TPC-DS queries. For example, RTFE queries own over six hundred of aggregations while TPC-DS queries have 40 aggregations at most, and many of the RTFE aggregations do not exist in TPC-DS queries (e.g., topN, last join, count_where). These observations indicate RTFE is richer in query operators and has much more complex operator

Table 3: Operator Comparison. The operators with underlines do not exist in TPC-DS and TPC-H.

	Top 10 most frequently used operators
FEBench	*last join*; *average*; *max*; *distinct count*; *top ratio*; *min*; *order by*; *(window) union*; *partition by*; *group by*
TPC-DS	*sum*; *case ...*; *count*; *distinct*; *with ... as*; *order by*; *left join*; *limit*; *group by*; *or*
TPC-H	*group by*; *order by*; *count*; *left join*; *average*; *case ...*; *having*; *with ... as*; *exists*; *min*



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

Figure 6: 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.

patterns than TPC-DS and TPC-H. Note analytical queries in HTAP benchmarks [25] have similar characteristics as OLAP queries, and so also cannot well handle the evaluation of RTFE systems.

Summary. Compared with existing database queries, feature extraction queries bring new challenges. (*i*) They involve some complex operators (e.g., *orderby/unions over long time windows*), which are uncommon cases in traditional database queries (Table 3). Note traditional stream queries are generally of simple operator patterns and do not involve *orderby* operators in online mode; (*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 (Figure 6); (*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 millisecond-level fraud detection.

6 BENCHMARK GENERATION

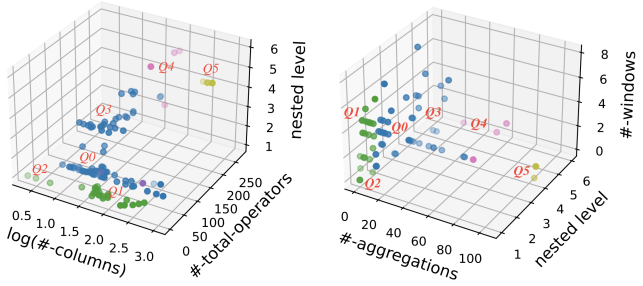
In this section, we first explain how to generate the FEBench benchmark by selecting templates out of the over 100 generated queries, some of which may contain similar query operators/structures, such that affecting the benchmark simplicity. Next we provide scenario analysis of these templates.

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 7).

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, customized aggregations), 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



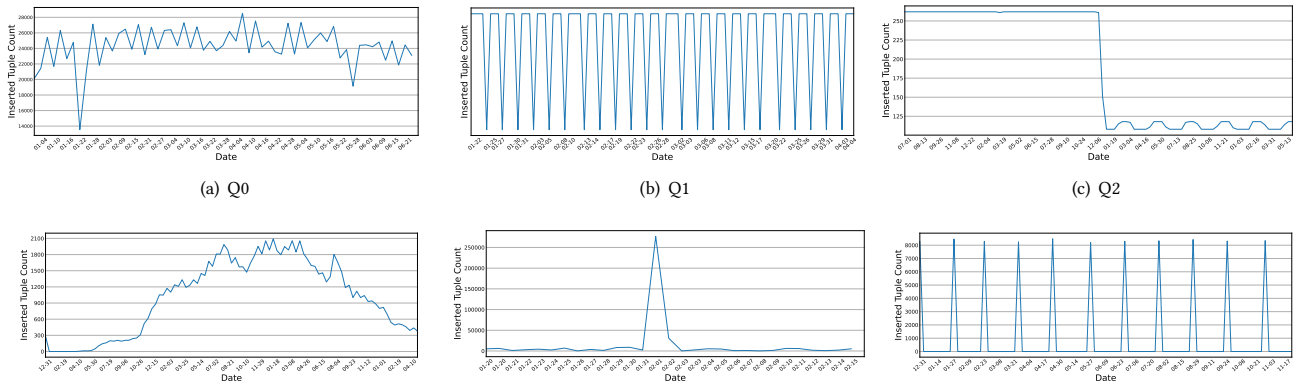
(a) High-level features (b) Highest-weighted features

Figure 7: Query Clustering Analysis. The 118 origin RTFE queries are divided into 6 clusters (query templates).

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 three 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 (note we iteratively adjust the epsilon distance and minimum number 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 7. 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 various 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 pick 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.* Note the first three templates are from public datasets, and the last three are real applications in the industry.



(a) Q0 (b) Q1 (c) Q2

(d) Q3 (e) Q4 (f) Q5

Figure 8: The real data distribution of selected six datasets in FEBench.

Table 4: The statistics of selected datasets.

Cluster	Task	Tables	Columns	Tuples
$Q0$	*Ride Prediction*	1	11	2.62×10^6
$Q1$	*Flu Forecast*	1	6	3.54×10^6
$Q2$	*Energy Forecast*	8	61	8.0×10^6
$Q3$	*Sales Prediction*	7	85	1.5×10^{10}
$Q4$	*Loan Evaluation*	9	245	1.0×10^9
$Q5$	*Fraud Detection*	10	773	1.3×10^{11}

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 *Q0-Q5*), we explain why they follow the four benchmark criteria (Section 3.4) by analyzing the data (Table 4) and query characteristics (Table 5).

6.2.1 Ride Duration Prediction (Q0). 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 massive GPS data of taxis and buses.

Data (6 months). For the ride prediction task, we collect open NYC transportation data ranging from 2016-01-01 to 2016-06-30. As shown in Figure 8 (a), the arrival time of the incoming data tuples is relatively random, with the number of inserted tuples within intervals ranging from 12,000 to over 28,000. The data schema consists of a single table with 11 columns, including two timestamp columns, two string columns, and seven numeric columns.

Query (single-table window). The features (output columns) in the query $Q0$ include three parts: (i) all the origin table columns (basic features); (ii) aggregation features separately from two time windows (e.g., distinct counts of "pickup_latitude" in last 1/2 hour); (iii) aggregation features computed across different time windows (e.g., the division of the 1h/2h window features). Since there is only one base table, $Q0$ is limited to *fundamental RTFE operators* (e.g., aggregations), and windows over the same table.

6.2.2 Flu Forecast (Q1). The task aims to predict the cumulative number of confirmed COVID19 cases in various places, as well as the number of resulting fatalities in the near future.

Data (2.5 months). For the flu forecast task, we have collected covid19 data ranging from 2020-01-22 to 2020-04-07. As shown in Figure 8 (b), the incoming data tuples follow a periodic distribution, i.e., in each cycle, the number of inserted tuples increases to the peak value (around 310). The data schema contains one base table with only 6 columns, including one timestamp columns, two string columns, three numeric columns. Note $Q1$ is not strict real-time (with day-level timestamps) and is taken as a special test case.

Query (variable-length windows). First, different from $Q0$, the query $Q1$ involves more window operators, e.g., there are 25 window-aggregation operators in $Q1$, while $Q0$ only has 11. The reason is that $Q1$ contains much fewer basic features (6 table columns) and it is vital to derive effective new features to support accurate forecast. Second, $Q1$ involves 10 different sizes of time windows. 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 *variable-length window processing* [22].

6.2.3 Wind Energy Forecasting (Q2). The task aims to predict the hourly power generation of the seven wind farms.

Data (3 years). For the power forecast task, we have collected the open wind-power data ranging from 2009-07-01 to 2012-06-26. Until Dec. 6th 2011, the number of incoming data stays steadily (around 265). After that, the number of incoming data sharply drops and periodically changes around 120. The data schema contains 10 tables with 61 columns, in which seven tables separately record the power changes in each farm (farm tables) and the rest three tables are used to train the prediction model (training tables).

Query (1:1 join across multiple tables). In the query $Q2$, joint features are computed from multiple tables (e.g., features of nearby farms and recent hours) by conducting the last join operator (Section 2.1) on 7 farm tables and 1 training table, providing a useful test for online one-to-one 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 numeric columns. Note that $Q2$ *does not contain time windows*, so all the historical tuples

Table 5: The statistics of selected query templates

	output columns / features	aggregations	last joins	time windows	subqueries	(window) unions	max window size
Q0	28	17	0	2	2	0	200
Q1	24	18	0	10	10	0	200
Q2	44	6	8	1	2	0	no limit
Q3	132	61	7	9	9	7	100
Q4	261	200	9	1	18	12	100
Q5	666	662	10	1	3	0	no limit

could contribute to the feature computation with incoming tuples, which is markedly different from traditional stream cases.

6.2.4 Sales Prediction (Q3). 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 real data in Uniqlo ranging from 2017-12-31 to 2021-05-30. Before May 30th 2018, data is rarely inserted. 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 per interval. And after Aug 4th 2020, the incoming data slightly decreases (over 300 tuples each day). The data schema owns 7 tables together with 85 columns, in which there are both steam tables, such as storing (canceled) orders, and attribute tables like product information.

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 up to 6-level subqueries that include both the whole tables (e.g., all tuples in 'product_item') and tables over time windows (e.g., sales within a 10-hour period). These types of subqueries can be costly to process due to the large amount of data involved and the computational overhead required for aggregating data over different time windows. Thus, $Q3$ is useful to test *the performance of complex subquery processing*.

6.2.5 Loan Evaluation (Q4). The task aims to predict whether customers will pay back their loans on time for a credit card company. $Q4$ 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 features from the origin tables 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 origin 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 $Q4$ can significantly affect the processing efficiency.

6.2.6 Fraud Detection (Q5). Fraud detection is a special type of the outlier detection problems, which aims to estimate the likelihood of fraudulent activities within milliseconds. Here we consider a fraud detection case in banking scenario. The temporal data was inserted periodically, with peak values around 8,000 in each cycle. $Q5$ owns 10 tables with over 13GB origin data and 773 columns. The query $Q5$ contains *the most query operators* among the 6 query templates, incorporating 659 RTFE operators, most of which are multi_last_value directly derived from the results of multi-table joins. Similar to $Q2$, the query of $Q5$ does not involve time windows, which leverages the entire dataset (all past user behaviors) to compute features for incoming tuples.

Summary. The benchmark satisfies the four criteria of a domain specific benchmark outlined by Jim Gray (Section 3.4). First, the query templates exhibit a diverse range of feature extraction patterns, e.g., the number of operators varies from 46 to 681, and the number of aggregations ranges from 11 to 652 (*relevance*). Second, each template is expressed in SQL-like language (*portability*) and incorporates unique operator patterns, i.e., (i) Q0-Q2 own relatively simple operators (on public datasets), but still surpass the complexity of traditional analytic queries (e.g., Q1 and Q2 involve dozens of aggregations and multi-table joins/windows) and (ii) Q3-Q5 own more tricky operator patterns (on real applications in 4Paradigm), which are difficult to optimize for ultra-low latency (*simplicity*). Third, each dataset can be scaled to larger or smaller sizes by following their real distribution of incoming data (*scalability*). We will highlight their respective strengths in evaluations.

7 EVALUATION OF DIFFERENT SYSTEMS

In this section, we introduce how to implement FEBench. As a starting point, we utilize FEBench to compare a general-purpose system (Flink [22]) and a specialized system (OpenMLDB [24]).

7.1 Overview of FEBench Pipeline

As shown in Figure 9, FEBench consists of three components: data loader, workload simulator and performance monitor. Taking product forecast as an example, *data loader* is responsible for loading static data such as the basic user/shop information into the system; *workload simulator* preloads all of the historical transactions (data tuples) into DRAM, and then sends the tuples one by one to simulate the arrival of new transactions. With each incoming transaction, the system performs a predefined RTFE query and inserts the new transaction. We record the response time of each transaction and reports performance metrics like 50th/99th/999th percentile latency [42] (denoted as TP-50/TP-99/TP-999). For simplicity, we define tail latency as TP-99/TP-999. FEBench is implemented in JMH (Java Microbenchmark Harness) [40], which communicates with different systems through Java clients.

7.2 Experiment Setting

System Setup We test two typical systems that support real-time feature extraction in SQL-like languages. First, Flink is a popular

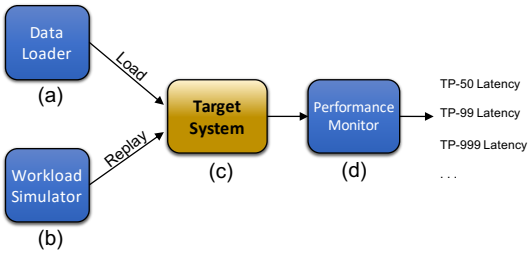


Figure 9: The overview of FEBench pipeline.

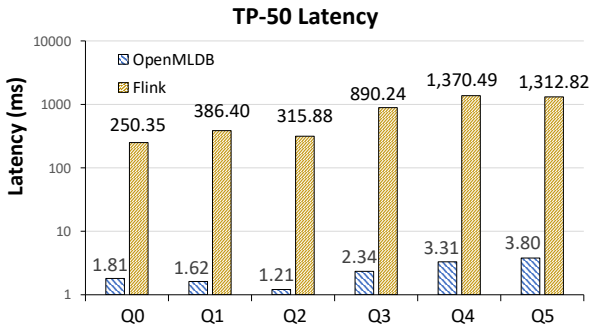


Figure 10: The TP-50 latency of all tasks (5 threads).

general-purpose stream platform. The version of Flink is 1.15.2, with one job manager node and three task manager nodes deployed as the Flink cluster. Flink caches part of the data in DRAM (the watermark mechanism is fine-tuned to ensure all the online required data is memory-cached) and persists data in RocksDB. Second, OpenMLDB is a popular specialized platform for real-time feature extraction. The version of OpenMLDB is 0.6.4, with three tablet servers and one name server deployed as the OpenMLDB cluster. OpenMLDB stores all the data in DRAM. To maintain data consistency, OpenMLDB writes logs on HDD for newly added data tuples. We plan to evaluate the benchmark for more systems such as Tecton in the future.

Hardware Setup We deploy both FEBench and the tested systems 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), 500 GB memory, and we use 7.3 TB hard disk as storage. The OS is CentOS-7.9 with kernel 3.10.0.

7.3 Overall Performance Results

In the following, we present some preliminary results and observations. *Our primary objective is not to compare the end-to-end performance of these systems. Instead, we demonstrate the usage of* FEBench *for illustrating the pipeline of our benchmark, and for showing the impact of different system implementation and design.*

Overall Performance. Different choices of implementation techniques can result in large performance gaps. As shown in Figure 10, the TP-50 latency of Q0/Q1/Q2 is shorter than that of Q3/Q4/Q5, as the latter queries require more complex operators (e.g., over 2x number of operators and more aggregations). Furthermore, Flink is almost two orders of magnitude slower than OpenMLDB. These performance differences are mainly due to implementation choices. Flink is implemented in Java to support various application scenarios and cross-platform deployment, and all RTFE queries are

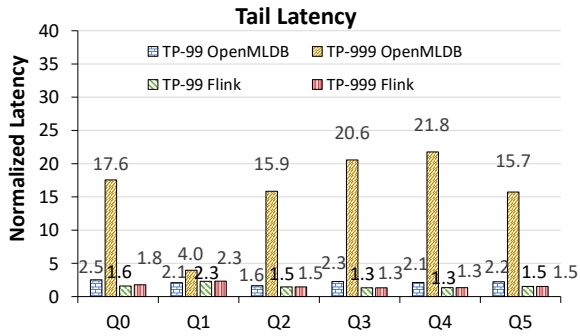


Figure 11: Normalized tail latency (all values are normalized to that of the TP-50 latency for each system).

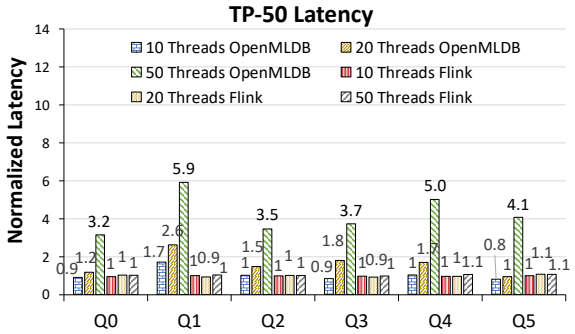


Figure 12: Normalized TP-50 latency (All values normalized to that of the 5 threads).

executed in the JVM. In contrast, OpenMLDB was initially designed for latency-sensitive applications like financial anti-fraud. Designed as a high-performance in-memory system, OpenMLDB uses Low-Level Virtual Machine (LLVM) [48] to transform RTFE queries into assembler code, which is much faster than that in the JVM [50].

Long Tail Latency. *OpenMLDB exhibits a noticeable long tail issue, while the tail latency of Flink is more stable.* To examine the growth rate of tail latency, we normalize the TP-999 and TP-99 latency to that of TP-50 latency for each system. As shown in Figure 11, the TP-999 latency of Flink only increases up to 12.69% compared to TP-99, whereas the TP-999 latency of OpenMLDB is up to 10.32 times that of TP-99 latency and up to 21.8 times that of TP-50 latency in Q0-Q5. Such long-tail issues have also been observed in other in-memory databases [24], which is mainly caused by the back-end `fsync` operations (for log writes) on the persistent storage. Instead, Flink stores newly added transaction data in a third-party database (e.g., RocksDB), which executes RTFE and new data inserts in different engines. Moreover, as discussed above, Flink takes longer time to complete each RTFE query, which reduces the frequency of new data insertions and hence mitigates the performance impact of database log writes.

Concurrency Performance. In Figure 12, latency values are normalized to that of five threads. We find OpenMLDB's TP-50 latency remains relatively stable when the thread number increases from five to twenty. However, setting the number of working threads to 50 leads to a significant increase in TP-50 latency (up to 5.92× that of five threads). In contrast, Flink exhibits smaller changes in TP-50 latency. There are two reasons. First, OpenMLDB strives to fully

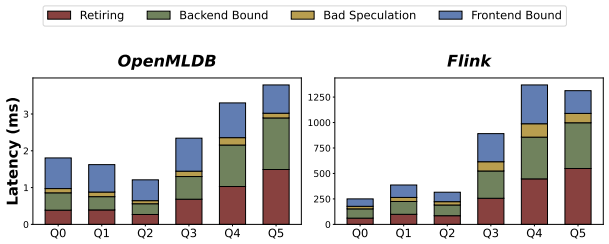


Figure 13: Micro-Architectural Metric Analysis

utilize system resources for each feature extraction request (such as ensuring high-parallelism computation for operators like aggregations), which may result in more frequent thread collisions in high-concurrency scenarios. Second, OpenMLDB already achieves millisecond-level latency, where even subtle performance degradation can be much more noticeable than in systems like Flink.

7.4 Profiling Analysis

Execution time breakdown We conduct the profiling study for detailed performance analysis and decompose the execution latency into TMAM metrics [49], including retiring (the time of retiring instructions), bad speculation (the time wasted due to branch prediction errors), backend_bound (instructions that cannot be dispatched due to resource shortage), frontend_bound (the time of fetching instructions and decoding them into executable micro-instructions).

As depicted in Figure 13, for Q0-Q2, the frontend_bound metric is most time-consuming (over 45%), where the operator patterns are relatively simple and most time is spent in switching between instructions of user request processing and feature extraction. This finding is consistent with the previous study on data streaming systems [51, 52]. For Q3-Q5, the most influential metrics are backend_bound (involving more secondary and intermediate tables that increase cache misses) and retiring (performing more instructions for complex operators like window unions). OpenMLDB reduces cache misses by reusing time windows over the same tables and performing customized operators (e.g., fetching the matched one tuple with most recent timestamp), which lead to superior performance in milliseconds. These query templates allow us to identify different performance bottlenecks at a micro-structural level.

Execution Plan. Next we use $Q0$ (ride duration prediction) as an example to illusrate the difference in execution plans. The execution plan of other queries could be found in our project website. The execution plan of $Q0$ includes three stages: (i) extract two time windows of taxi X, and execute aggregations, (ii) extract basic information of taxi X and execute calculations on the selected columns, (iii) join the results of the first two parts and execute calculations to output results. As shown in Figure 14, a series of operators have been adopted, including *request* (locate the required data), *exchange* (pass the intermediate results to other nodes), *agg:xh* (aggregation on x-hour window), *limit* (truncate data), *join* (concatenate the intermediate results), and *calc* (calculate on the input data).

We have the following observations. First, in Flink, the *calc* operator takes the most time and there are 28 features to compute in Q0, whose computation complexity is much higher than traditional stream queries. Instead, OpenMLDB spends most time on initial

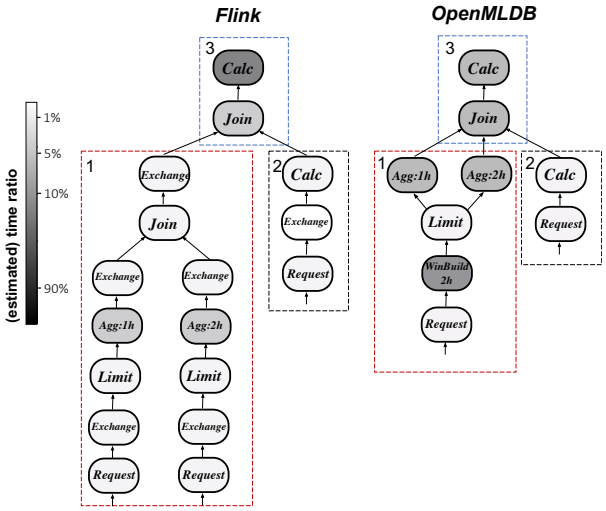


Figure 14: Example Execution Plans (Q0). Each dotted box represents a segment in the query plan.

window computation, whose results will be reused by the following window operators. Second, to reduce the latency of each RTFE query, OpenMLDB has made several optimizations at plan level. On one hand, $Q0$ requests to read two time windows (i.e., past 1/2 hours of taxi X). The default behavior of the plan is to read the data of two time windows separately. Instead, OpenMLDB senses the overlap of the time windows, reads only the data of the larger 2-hour time window, and performs the aggregation operator on the 1-hour and 2-hour time windows respectively, which reduces the duplicate data reading overhead and leads to lower retiring (Figure 13). On the other, OpenMLDB adopts multiple optimized operators to reduce the latency: (i) the customized aggregation functions (e.g., distinct_count, count_where) help to streamline the required instructions (lower retirng); (ii) the lightweight join operator (*last join* in Section 2.1) matches tuples on the index of the data, avoids the duplication check (only one result tuple) and data copy overhead, and can efficiently execute within limited memory space (lower backend_bound).

8 CONCLUSION AND FUTURE WORK

Real-time feature extraction is an emerging trend and widely taken as essential to enable AI applications in production. In this paper, we first explained how to borrow the ideas in relational data and SQL to conduct feature extraction for real-world applications. Next, based on the over 100 collected real datasets, we proposed a benchmarking architecture FEBench for real-time feature extraction, involving dataset collection, workload analysis, template generation, and system deployment. The preliminary results showed that FEBench can effectively reflect the strengths and weaknesses of both the general-purpose system (Flink) and specialized system (OpenMLDB). More information could be found in our project site: *https://github.com/decis-bench/febench.*

ACKNOWLEDGEMENTS

This paper was supported by 4Paradigm, NSF of China (61925205, 62232009, 62102215), Huawei, TAL education, and BNRist.

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.tecton.ai/.
[12] https://www.tpc.org.
[13] https://www.tpc.org/tpcds/.
[14] https://www.tpc.org/tpch/.
[15] Forecast: The business value of artificial intelligence. In *Gartner*, 2018.
[16] R. Ahmed, A. W. Lee, A. Witkowski, D. Das, H. Su, M. Zaït, and T. Cruanes. Cost-based query transformation in oracle. In *Proc. VLDB Endow.*, pages 1026–1036. ACM, 2006.
[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] 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.
[27] 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.
[28] D. S. Evans. The online advertising industry: Economics, evolution, and privacy. *Journal of economic perspectives*, 23(3):37–60, 2009.
[29] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (1st Edition)*. Morgan Kaufmann, 1991.
[30] I. Guyon, L. Sun-Hosoya, M. Boullé, H. J. Escalante, S. Escalera, Z. Liu, D. Jajetic, B. Ray, M. Saeed, M. Sebag, et al. Analysis of the automl challenge series. *Automated Machine Learning*, 177, 2019.
[31] M. A. Hall. *Correlation-based feature selection for machine learning*. PhD thesis, The University of Waikato, 1999.
[32] M. A. Hall and L. A. Smith. Practical feature subset selection for machine learning. 1998.
[33] S. Hur and J. Kim. A survey on feature store. *Electronics and Telecommunications Trends*, 36(2):65–74, 2021.
[34] 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.
[35] 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.
[36] I. Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.
[37] 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.
[38] G. Li, X. Zhou, and L. Cao. AI meets database: AI4DB and DB4AI. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2859–2866. ACM, 2021.
[39] 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.
[40] OpenJDK, 2013. https://openjdk.java.net/projects/code-tools/jmh/, Last accessed on 2020-11-15.
[41] 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.
[42] T. percentile. Tp-x. https://support.huaweicloud.com/intl/en-us/productdesc-apm/apm_06_0002.html, 2019.
[43] V. Steinbiss, B.-H. Tran, and H. Ney. Improvements in beam search. In *Third international conference on spoken language processing*, 1994.
[44] C. Sun, N. Azari, and C. Turakhia. Gallery: A machine learning model management system at uber. In *EDBT*, pages 474–485, 2020.
[45] Y. Tay. Data generation for application-specific benchmarking. *Proceedings of the VLDB Endowment*, 4(12):1470–1473, 2011.
[46] T. Tsai. Competitive landscape: Ai startups in china. In *Technical Report*.
[47] 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.
[48] Wikipedia. *LLVM*, 2019. [Online; accessed 02-July-2022].
[49] A. Yasin. A top-down method for performance analysis and counters architecture. In *ISPASS*, pages 35–44. IEEE Computer Society, 2014.
[50] 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.
[51] 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 *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 659–670, 2017.
[52] S. Zhang, J. He, A. C. Zhou, and B. He. Briskstream: Scaling data stream processing on shared-memory multicore architectures. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 705–722, New York, NY, USA, 2019. Association for Computing Machinery.
[53] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(3):1096–1116, 2022.
[54] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *Proc. VLDB Endow.*, 15(1):46–58, 2021.