A Case for Enrichment in Data Management Systems

Dhrubajyoti Ghosh[1], Peeyush Gupta[1], Sharad Mehrotra[1], and Shantanu Sharma[2]
[1]University of California, Irvine, USA. [2]New Jersey Institute of Technology, USA.

ABSTRACT

We describe ENRICHDB, a new DBMS technology designed for emerging domains (*e.g.*, social media analytics and sensor-driven smart spaces) that require incoming data to be enriched using expensive functions prior to its usage. To support online processing, today, such enrichment is performed outside of DBMSs, as a static data processing workflow prior to its ingestion into a DBMS. Such a strategy could result in a significant delay from the time when data arrives and when it is enriched and ingested into the DBMS, especially when the enrichment complexity is high. Also, enriching at ingestion could result in wastage of resources, if applications do not use/require all data to be enriched. ENRICHDB's design represents a significant departure from the above, where we explore seamless integration of data enrichment all through the data processing pipeline — at ingestion, triggered based on events in the background, and progressively during query processing. The cornerstone of ENRICHDB is a powerful *enrichment data and query model* that encapsulates enrichment as an operator inside a DBMS enabling it to co-optimize enrichment with query processing. This paper describes this data model and provides a summary of the system implementation.

1. INTRODUCTION

This paper envisions a new type of data management technology that seamlessly integrates *data enrichment* in the data analysis pipeline. By data analysis pipeline, we refer to the process of acquiring data from data sources, potentially enhancing the data, ingesting it into a database system, and running queries on the enhanced data. Today, organizations have access to potentially limitless data sources in the form of web data repositories, social media posts, and continuously generated sensory data. Such data is often low-level/raw and needs to be enriched to be useful for analysis. Functions used to enrich data (referred to as *enrichment functions* in the paper) could consist of (a combination of) custom-compiled code, declarative queries, and/or expensive machine learning techniques. Examples of enrichment functions include mechanisms for sentiment analysis over social media posts, named entity extraction in text, and sensor interpretation and fusion over sensory inputs.

Traditionally, data enrichment is performed offline as part of a periodic Extract-Transform-Load (ETL) process. This process is performed inside a separate system and the enriched data is stored in a data warehouse for analysis. This approach adds significant latency between the time data arrives (or is created) and when it is available for analysis.

[25, 20] have highlighted limitations of traditional data warehouses approach in analyzing the recent data (as it arrives) in the context of online business applications. It has led to the emergence of Hybrid Transaction/Analytical Processing (HTAP) systems that support both transactional and analytical workloads. A warehouse strategy (of periodic enrichment as part of ETL) exhibits similar limitations in application contexts, where enrichment is part of the data processing pipeline. One possibility to overcome this limitation is enriching the data as it arrives. Systems (*e.g.*, Spark Streaming [31] often used for scalable ingestion) are capable of executing enrichment functions on newly arriving data prior to its storage in a DBMS. Recently, [28] has explored ways to optimize enrichment during ingestion by batching such operations.

Enriching data at arrival is only feasible when enrichment functions are simple. Complex functions (*e.g.*, Multi-layer Perceptron and Random Forest), often, used to classify/interpret incoming data, may take in the order of several hundred milliseconds to execute on a single core of a modern server.[1] Applying such functions at ingestion will allow a system to ingest only tens of events per second per core which is very low. In our experience, enriching WiFi connectivity data (using predictive models [19]) for fine-grained localization to support location-based services is simply infeasible to scale to campus-level, if we enrich data as it arrives (arrival rate of 1000 events/sec and takes 200 ms/event to process).[2]

An alternate strategy is to restrict ETL process to selectively enrich only a part of the data (based on expected usage) at ingestion. However, predicting usage is difficult, especially in an online setting where an analyst can pose any adhoc query. If the prediction underestimates the need of enrichment, it may not support certain queries and overestimation leads to wasted enrichment and resources.

Another weakness of enrichment at data ingestion (both completely and selectively) is non-adaptiveness. It requires analysts to pre-specify the set of enrichments that needs to be performed on the data for future data accesses. If requirements of an analyst change and/or new enrichment functions are added, it will require re-execution of ETL pipeline.

[1] *E.g.*, a server of 64 core Intel Xeon CPU E5-4640, 2.40GHz, and 128GB memory.

[2] Similar scenarios exist in social media and multi-media domains where a classifier of Decision Tree takes 13 ms, Naive Bayes takes 110 ms, and a Support Vector Machine takes 800 ms per image to classify objects present in it.

Motivated by the above-mentioned limitations, we design EnrichDB — an adaptive data management technology that allows enrichment to be performed all through the data processing pipeline, *i.e*., during ingestion, triggered based on events, or during query processing. EnrichDB is designed based on the following criteria:

Semantic Abstraction. EnrichDB supports a declarative interface to specify and link enrichment functions with higher-level observations that the functions generate from raw data. Users may associate one or more such functions that differ in terms of quality (*e.g*., uncertainty in the enriched value) and cost (*e.g*., execution time of the function). In EnrichDB, developers do not need to deal with raw data directly — applications can be fully written based on higher-level semantic observations derived from raw data using enrichment functions. If the higher-level observation has not been derived through the enrichment process prior to the execution of the application, such enrichment would be automatically performed as part of query processing.

Transparency of Enrichment. In EnrichDB, the data enrichment process is transparent to application programmers, who view the data at a semantically higher level of abstraction. Programmers do not need to be concerned about what data needs to be enriched, using which functions, and at what stage of data processing. EnrichDB maintains the state of enrichment of objects, performs enrichment automatically based on the current state and needs of applications, and updates the state appropriately.

Optimization of Enrichment. In order to mitigate the ingestion latency due to complex functions, EnrichDB allows enrichment all through the data processing pipeline. Enrichment can be performed at ingestion, triggered based on events, or during query processing. EnrichDB makes sure that enrichment of objects is performed optimally. During query time enrichment, EnrichDB exploits the query optimizer to prune away enrichment of objects that do not influence the query results. The developer does not have to write any separate code to prune such enrichment of objects. Furthermore, EnrichDB allows enrichment of data closer to where the data resides resulting in a low data movement.

Progressive Computation. EnrichDB produces answers progressively, as it executes enrichment functions as a part of query processing. A progressive query answering (motivated by Approximate Query Processing systems [14] — provided progressive query answering for aggregation queries) technique produces an initial set of answers that are improved over time as data is further enriched.

The cornerstone of EnrichDB is *Enrichment Data and Query Model* (EDQM) that integrates enrichment as a first-class operator in the database system. This paper describes both data and query models in §2 and briefly describes the implementation of EnrichDB in §3.

2. DATA AND QUERY MODEL

In this section, we develop a new data and query model, called Enrichment Data and Query Model (EDQM).

2.1 Data Model

In EDQM, the data is modeled using relations where a relation can have two types of attributes: (*i*) ***derived*** attributes that require enrichment and (*ii*) ***fixed*** attributes that do not require enrichment. Each derived attribute is optionally associated with a domain size. If the domain size is not specified, then that attribute is considered to have a value from a continuous range. The command for specifying a relation in EnrichDB is shown below.

```
CREATE TABLE TweetData(tid char(8), userid
   char(20), Tweet text,feature float[], topic
   int derived:40, sentiment int derived:3,
   TweetTime timestamp,location text);
```

The value of a derived attribute is determined using one or more ***enrichment functions*** associated with it.

Enrichment functions. EDQM supports a general class of enrichment functions (frequently used in real-world). The input to an enrichment function is a tuple and the output is either a single value, multiple values, or a probability distribution, as described below.

We categorize enrichment functions based on the output cardinality: (*i*) *single-valued*: outputting a single value, *e.g*., a binary classifier [27], (*ii*) *multi-valued*: outputting a set of values as a prediction, *e.g*., top-k classifiers [18], (*iii*) *probabilistic*: outputting a probability distribution over the possible values of a label, *e.g*., probabilistic classifiers [10]. Also, enrichment functions can be categorized based on the output domain size: (*i*) *categorical*: predicts outputs from a finite set of possible values, *e.g*., sentiment of positive/negative, and (*ii*) *continuous*: outputs a real number, *e.g*., a weather of 72.8°F.

An enrichment function is associated with two parameters: (*i*) *cost*: the average execution time/tuple, and (*ii*) *quality*: a metric of the goodness (*i.e*., accuracy) of enrichment function in determining the correct value of the derived attribute.

Training of enrichment functions. EDQM supports training procedures for enrichment functions where a user needs to specify an input table storing the training data. Below, we show an example of learning a machine learning model of Multi-Layer Perceptron (MLP) using a training procedure of `model_train`. This model is trained using data stored in `tweets_train` table and the name of the model is `sentiment_mlp`. It uses attribute values of `feature` as input to the model and outputs prediction for `sentiment` attribute. The model-specific parameters are passed as a string in `model_params`.

```
SELECT db.model_train(
   'tweets_train', 'sentiment_mlp',
   'multi_layered_perceptron','sentiment',
   'feature', 'classification', model_params);
```

tid	UserID	Tweet	feature	loc.	TweetTime	**topic**	**sentiment**
t_1	John	Upload ...	[0.2,...,0.4]	US	16:08	soc	pos.
t_2	Mark	Listening...	[0.5,...,0.3]	US	16:48	ent.	NULL
t_3	Richard	Iran's ...	[0.6,...,0.4]	UK	11:48	pol.	neg.

Table 1: `TweetData` table in ENRICHDB. Derived attributes are **`topic`** and **`sentiment`** and the values are their determinized representations.

tid	topic	sentiment
t_1	soc:0.54, ent:0.46	pos:0.52, neu:0.48
t_2	ent: 0.65, art: 0.35	pos:0.5,neu:0.5
t_3	pol:0.8, art: 0.2	neu:0.3,neg:0.7

Table 2: State output for derived attributes.

The *cost* and *quality* of enrichment functions can either be specified by the user or can be determined automatically by using several methods, *e.g.*, train/test split, k-fold cross-validation, and leave-one-out cross-validation, during the training phase. The set of enrichment functions for a derived attribute $\mathcal{A}_i$ are called ***function-family*** of $\mathcal{A}_i$. (We use ***calligraphic font*** for ***derived attributes***.) Outputs of enrichment functions in a function-family are combined using a ***combiner function***. One can use weighted-average, majority-voting, or stacking-based [30] combiner functions in ENRICHDB. As an example shown below, the function-family of `sentiment` attribute is created using an `assign_enrichment_function` function. It uses `mlp_classifier` function with *cost* of 0.1 second/tuple and *quality* of 0.8 (measured in AUC).

```
SELECT db.assign_enrichment_function
  ('sentiment_fmly','TweetData', 'sentiment',
  ['mlp_classifier','sentiment_mlp',0.1,0.8]);
```

State of a Derived Attribute. Enrichment state or state of a derived attribute $\mathcal{A}_i$ in tuple t_k (denoted by $state(t_k.\mathcal{A}_i)$) is the information about enrichment functions that have been executed on t_k to derive $\mathcal{A}_i$. The state has two components: ***state-bitmap*** that stores the list of enrichment functions already executed on $t_k.\mathcal{A}_i$; and ***state-output*** that stores the output of executed enrichment functions on $t_k.\mathcal{A}_i$. *E.g.*, consider that there are four enrichment functions f_1,f_2,f_3,f_4, and out of which f_1,f_3 have been executed on $t_k.\mathcal{A}_i$. Also, assume that the domain of $\mathcal{A}_i$ contains three possible values: d_1, d_2, and d_3. Thus, the state-bitmap for $t_k.\mathcal{A}_i$ contains $\langle 1010\rangle$, *i.e.*, only first and third functions are executed and the state-output of $t_k.\mathcal{A}_i$ contains: $\langle$[0.7,0.3,0],[],[0.8,0.1,0.1],[]$\rangle$, *i.e.*, the output of the first and third enrichment functions (remaining arrays are left empty).

The state-output stores a list of probability distributions when the enrichment functions are probabilistic classifiers, *e.g.*, $\langle[0.7, 0.3, 0], [], [0.8, 0.1, 0.1], []\rangle$. For single-valued classifiers, clustering functions, and regression functions the state-output attribute stores the actual output of the function instead of a probability distribution, *e.g.*, $\langle$[72.4],[],[76.8],[]$\rangle$.

State of Tuples and Relations. The notion of state of derived attributes is generalized to the state of tuples and relations in a straightforward way. The state of a tuple t_k is the concatenation of the state of all derived attributes of t_k, *e.g.*, the state of a tuple t_k of a relation R with three derived attributes $\mathcal{A}_p$, $\mathcal{A}_q$, and $\mathcal{A}_r$ is denoted by $state(t_k) = \langle state(t_k.\mathcal{A}_p)||state(t_k.\mathcal{A}_q)||state(t_k.\mathcal{A}_r)\rangle$.

Relative Ordering of Enrichment Functions. In EDQM, the user can specify (or can be learned by ENRICHDB using a training dataset) the relative order in which enrichment functions need to be executed. This order is specified using the state of tuples for each derived attribute. Such relative ordering is important for ensembling different enrichment functions to be executed on a tuple. This ordering is stored in a table called `DecisionTable`.

This table, for each derived attribute of a relation, stores a map that — given the current state of a tuple with respect to the attribute — specifies the next function that should be executed to further enrich the attribute, as well as (optionally) the measure of ***benefit*** that is expected to result. The details of how to use this table in ENRICHDB and how to learn it based on a training dataset are presented in §3.1.

2.2 Query Model

This section describes the query language of ENRICHDB (§2.2.1), query semantics (§2.2.2), and the goal of enrichment (§2.2.3) in the context of a given query in ENRICHDB.

2.2.1 Query Language

The query language of ENRICHDB is an extended version of SQL. Queries in ENRICHDB are associated with mandatory query semantics (which are required to deal with probabilistic values of derived attributes) and a (optional) quality parameter for the quality of the query results.

Two types of query semantics for probabilistic data have been proposed: (*i*) determinization-based semantics [11] and (*ii*) possible world (PW) semantics [26]. The determinization-based semantics converts probabilistic representation to a single or a small set of deterministic worlds. The query is executed in these worlds and a single deterministic answer is produced. In contrast, in PW semantics, all possible worlds are generated (implicitly/explicitly) from probabilistic representation and the query is executed in each world. The result consists of all possible tuples along with their probability of being part of the result in at-least one world. The rationale of choosing one semantics over the other depends on the application scenarios. In some scenarios, an application can make good decisions by just using the most probable answers, whereas for some applications, it may require analysis of all possible answers along with their probability distribution. Due to simplicity, we have implemented the determinization-based query semantics in ENRICHDB (the implementation of PW semantics is under development).

An example query in ENRICHDB is shown below:

```
SELECT Tweet,location,TweetTime, topic,
```

C_1	T	F	P	P	P	P	U
C_2	P	P	T	F	P	U	P
$C_1 \wedge C_2$	P	F	P	F	P	U	U
$C_1 \vee C_2$	T	P	T	P	P	P	P
NOT C_1	F	T	F	F	F	F	U

Table 3: Truth table for evaluating complex conditions.

```
sentiment FROM TweetData WHERE sentiment='pos'
AND topic='soc. media' AND TweetTime
BETWEEN('16:00','18:00') QUALITY 0.9
SEMANTICS DETERMINIZATION/PROBABILISTIC;
```

The `QUALITY` and `SEMANTICS` keywords specify the minimum quality requirement of query result and the semantics of query evaluation respectively.

2.2.2 Determinization-Based Query Semantics

In determinization-based query semantics, tuples of all participating relations in a query are determinized first before evaluating the query. The process of converting a probabilistic data representation, *i.e.*, the output of probabilistic enrichment functions, to a deterministic representation is referred to as the *determinization process*.

Consider a derived attribute $\mathcal{A}_i$ and a tuple t_k. The value of tuple t_k in attribute $\mathcal{A}_k$ (*i.e.*, $t_k.\mathcal{A}_i$) is determined using a *determinization function* ($DET(.)$) based on tuple's state. $DET(state(t_k.\mathcal{A}_i))$ returns a single or multiple values for $t_k.\mathcal{A}_i$ or a NULL value, representing a situation when state of the attribute does not provide enough evidence to assign any value for $t_k.\mathcal{A}_i$. Determinization concept naturally extends to a tuple and a relation. The determinized representation of a relation R is denoted by:

$$DET(R) = DET(state(t_i.\mathcal{A}_j)) \,|\, \forall t_i \in R, \forall \mathcal{A}_j \text{ of } R.$$

Example. Consider a relation `TweetData` with two derived attributes `topic` and `sentiment` (see Table 1), and assume state-output of derived attributes as shown in Table 2. Based on Table 2 and a top-1 determinization strategy, the determinized representation of `topic` and `sentiment` are shown in Table 1. ■

Since determinization of a tuple can result in either a set of values or NULL, evaluation logic of different conditions needs to be defined. EnrichDB extends the traditional three-valued logic used in relational operators, *i.e.*, with truth values of $\boldsymbol{T}$, $\boldsymbol{F}$, and $\boldsymbol{U}$ into a four-valued logic: **true ($\boldsymbol{T}$)**, **false ($\boldsymbol{F}$)**, **possible ($\boldsymbol{P}$)**, and **unknown ($\boldsymbol{U}$)**. Here, P represents that the condition is **possibly true** based on the current state of enrichment, whereas U (as in traditional setting) represents that the truth value is **unknown**, given the current level of enrichment. Similar to SQL, the DBMS implementing this data model does not have to return tuples that evaluate to unknown. However, the tuples evaluating to **possible** may or may not be returned. *E.g.*, the inclusion of such tuples in the answer could be based on the maximization of the quality of the query. We next discuss how we assign truth values to predicates/expressions.

Simple Predicates. Consider an expression $\boldsymbol{\mathcal{A}_i}$ **op** $\boldsymbol{a_m}$, where $\mathcal{A}_i$ is a derived attribute, `op` is an operator, and a_m is a possible value of $\mathcal{A}_i$. The operator `op` is one of the following operators: $\langle =, \neq, >, \geq, <, \leq \rangle$. If the output of $DET(state(t_k.\mathcal{A}_i))$ is NULL, then the expression evaluates to U. If $DET(state(t_k.\mathcal{A}_i))$ is a singleton set S and $x \in S$ such that x `op` a_m holds, then the expression evaluates to T; otherwise, F. If $DET(state(t_k.\mathcal{A}_i))$ is a multi-valued set (say S) and $\exists x \in S$ such that x `op` a_m holds, then it is possible that t_k satisfies the expression, and hence, it evaluates to P. However, if $\nexists x \in S$ for which x `op` a_m holds, then the expression evaluates to F.

Consider an expression $\boldsymbol{\mathcal{A}_i}$ **op** $\boldsymbol{\mathcal{A}_j}$, where $\mathcal{A}_i$ and $\mathcal{A}_j$ are two derived attributes of (possibly different) relations and `op` is a comparison operator. If $DET(state(t_k.\mathcal{A}_i))$ or $DET(state(t_l.\mathcal{A}_j))$ is NULL, then the condition evaluates to U. If both $DET(state(t_k.\mathcal{A}_i))$ and $DET(state(t_l.\mathcal{A}_j))$ are singleton sets and for elements $x \in DET(state(t_k.\mathcal{A}_i))$ and $y \in DET(state(t_l.\mathcal{A}_j))$, x `op` y holds, then the condition evaluates to T; otherwise, F. In case one or both of $DET(state(t_k.\mathcal{A}_i))$ and $DET(state(t_l.\mathcal{A}_j))$ are multi-valued sets and $\exists x \in DET(state(t_k.\mathcal{A}_i))$ and $\exists y \in DET(state(t_l.\mathcal{A}_j))$, such that x `op` y holds, then the condition evaluates to P; otherwise, F.

Complex Predicates. Complex predicates are formed using multiple comparison conditions connected by Boolean operators (AND ($\wedge$), OR ($\vee$), and NOT ($\neg$)). Table 3 shows the truth table for such logical operators. This table only shows entries when one of the two expressions evaluates to P. When both expressions evaluate to either T, F, or U, we follow the same evaluation logic as in standard SQL.

Aggregation. Aggregation functions on fixed attributes are evaluated as in SQL, while, on a derived attribute, return a range of values $[l,u]$, denoting the lower and upper bounds of aggregated value. An aggregation function (*e.g.*, *count*, *sum*, *min*) applied to all T tuples of a set produces the value of lower bound l, while applied to all T and P tuples together produces the upper bound u. *E.g.*, consider the query of §2.2 on Table 1, and assume that the table has 250 tuples of which 100 tuples evaluate to T, while 20 of the remaining 150 tuples evaluate to P. Hence, the condition evaluation logic returns a range of $[100, 120]$. Likewise, group-by aggregation results in one such range per group.

Top-k Aggregation. EnrichDB first evaluates aggregation functions for each group-by key (as described above), and their outputs are ranked using a ranking function. The query result consists of a set of group-by keys with the top-k ranks. The purpose of the ranking function is to return a minimal answer set A, such that the real top-k groups are guaranteed to be part of A. EnrichDB sorts the group-by keys based on the lower bounds in a descending order and selects the first n (where $n \geq k$) group-by keys as the minimal answer set A such that the upper bound of $(n+1)$-th key is lower than the lower bound of the n-th key. This ensures that the $(n+1)$-th group-by key cannot be part of the top-k answer set.

Consider a query that returns top-2 topics with the highest tweet counts from Table 1. Suppose after applying $count()$, the topics had following bounds: social media: [100,150], entertainment: [110,120], politics: [100,115], and economy: $[80, 95]$. The answer of the query will be {social media, entertainment, politics} that guarantees that the actual top-2 groups (*i.e.*, social media and entertainment) are part of the answer. The group economy will be excluded from the answer, since the upper bound of this group is 95, which is lower than the lower bounds of groups inside the answers.

Query Semantics. Now, based on the definition of determinization function and the predicate evaluation logic as described above, we define the query semantics as follows: $q(R_1,R_2,...,R_n) = q'(DET(R_1),DET(R_2),...,DET(R_n))$

Here, $q(R_1,R_2,...R_n)$ is a query on relations $R_1,...,R_n$, $DET(R_i)$ is the determinized representation of the i^{th} relation. Query q is rewritten as q' to be executed on the determinized representations of relations using the four valued logic as described above.

2.2.3 Quality Measure of Query Results

In determinization-based query semantics, we measure the quality of answers to (*i*) set based queries using Jaccard's similarity or expected F_α-measure, (*ii*) aggregation queries using the root-mean-square error, mean absolute error, or the half-interval length of query answer, and (*iii*) group-by and top-k queries using the summation of half-interval lengths of all group by keys. In contrast, in PW semantics, we measure the quality of answers to both set-based and aggregation queries using entropy of the returned query result [9].

Progressive Query Execution. For producing query answers in epochs, ENRICHDB supports a mechanism to refine previously produced query answers, by retracting or adding tuples for set-based results or by improving the confidence intervals [14] for aggregation results. Specifically, the answer set $Ans(q,e_k)$ for a query q at the end of an epoch e_k is calculated as follows:

$$Ans(q,e_k) = \{Ans(q,e_{k-1}) \cup \Delta(q,e_k)\} \setminus \nabla(q,e_k) \quad (1)$$

where $\Delta(q,e_k)$ ($\nabla(q,e_k)$) is the set of tuples added to (removed from) query answers of epoch e_{k-1} at epoch e_k. We refer to both these sets as ***delta answers***.

The notion of answer modification using deletion and addition of tuples generalizes to aggregation queries. For aggregation queries, the result contains a tuple for each `GROUP BY` key, where the tuple consists of an aggregated value. In epoch e_k, if the aggregated value changes for a `GROUP BY` key, then the corresponding tuple in the result of previous epoch e_{k-1} is deleted and a new tuple with the updated value is added.

Progressive Score. Since ENRICHDB allows users to stop query evaluation at any instance of time (even before the quality requirement is met), performing enrichments impacting answer quality as early as possible is needed. ENRICHDB's effectiveness is measured using the following progressive score (similar to [22, 6]):

$$\mathcal{PS}(Ans(q,E)) = \sum_{i=1}^{|E|} W(e_i) \cdot [\mathcal{Q}(Ans(q,e_i)) - \mathcal{Q}(Ans(q,e_{i-1}))]$$

The query execution time is discretized into sub-intervals, called *epochs* ($\{e_1, e_2, ..., e_z\}$), $W(e_i) \in [0,1]$ is the weight allotted to the epoch e_i, $W(e_i) > W(e_{i-1})$), $\mathcal{Q}$ is the quality of answers, and $[\mathcal{Q}(Ans(q,e_i)) - \mathcal{Q}(Ans(q,e_{i-1}))]$ is the improvement in the quality of answers occurred in the epoch e_i. The quality $\mathcal{Q}$ is measured according to the type and semantics of the query as discussed above. Given a query, a quality requirement, and a set of weights assigned to epochs, ENRICHDB's goal is to achieve maximum progressive score for the query, if query execution is stopped early.

3. ENRICHDB IMPLEMENTATION

There are two possible ways of implementing the above data model as (see Figure 1): (*i*) a *loosely coupled* (LC) approach, wherein an enrichment module is implemented separately from DBMS, and (*ii*) a *tightly coupled* (TC) approach, wherein an enrichment module is tightly integrated with a query processing module of DBMS. ENRICHDB follows TC approach as it offers several benefits as discussed below.

The TC approach uses query context to eliminate redundant enrichment. *E.g.*, consider the query of §2.2 that retrieves tweets with `sentiment='positive' AND topic='social media'`. Say values of both t.`topic` and t.`sentiment` were NULL for a tuple t. In LC approach, all the functions associated with `sentiment` and `topic` would be used for enriching t. In contrast, the TC approach of ENRICHDB uses a feedback-based mechanism from the query engine to direct the enrichment process. In particular, if a tuple t does not meet the condition of `sentiment='positive'`, then that tuple is not enriched using topic-based enrichment functions. Such a pruning strategy can be very effective, when queries are complex and selective. Also, the TC approach executes enrichment functions closer to data (in the database engine), and, allows us to incorporate both determinization and possible world-based query semantics.

ENRICHDB is implemented using TC approach on top of PostgreSQL. An insert command for adding data to ENRICHDB, is transformed into a PostgreSQL insert command. An ENRICHDB query is wrapped in a stored procedure that internally executes appropriate SQL queries on top of PostgreSQL tables during multiple epochs. The query results are maintained using Incremental Materialized Views (IMV) [3] to reduce the overhead of executing queries multiple times. Enrichment functions are implemented as user-defined functions (UDFs), and their execution is orchestrated by a special UDF that executes enrichment function as UDFs by taking them as arguments.

3.1 Schema Manager

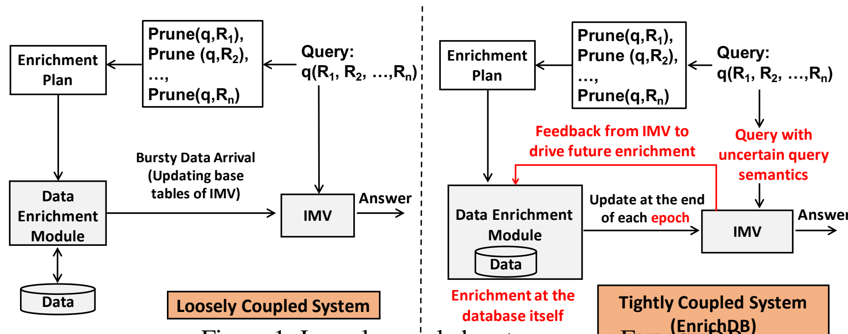

Figure 1: Loosely coupled system versus EnrichDB.

Table	attribute	type	NumLabel	StateCutoff
TweetData	tid	fixed	N/A	N/A
TweetData	UserID	fixed	N/A	N/A
TweetData	TweetTime	fixed	N/A	N/A
		...		
TweetData	topic	derived	40	True
TweetData	sentiment	derived	3	False

Table 4: `RelationMetadata`.

FamilyID	FID	AttributeName	RelationName
ff_1	f_1	sentiment	TweetData
ff_1	f_2	sentiment	TweetData
ff_1	f_3	sentiment	TweetData
ff_2	f_4	topic	TweetData
ff_2	f_5	topic	TweetData
ff_2	f_6	topic	TweetData

Table 5: `FunctionFamily`.

FID	Function	ModelTable	InputColumn	OutputColumn	params	cost	quality
f_1	*mlp_classifier()*	*sentiment_mlp_model*	feature	sentiment	N/A	0.16	0.80
f_2	*dt_classifier()*	*sentiment_dt_model*	feature	sentiment	N/A	0.08	0.72
			...				
f_6	*dt_classifier*	N/A	feature	topic	topic/decisiontree	0.12	0.68

Table 6: `FunctionMetadata`, storing the metadata of enrichment functions.

RelationName	AttributeName	Map
TweetData	sentiment	$\langle 1,0,0\rangle$,[0-0.25) : $\langle f_2,0.1\rangle$, $\langle 1,0,0\rangle$,(0.25-0.5) : $\langle f_3,0.2\rangle$, $\langle 1,0,0\rangle$,(0.5-0.75) : $\langle f_2,0.16\rangle$,$\langle 1,0,0\rangle$,(0.75-1] : $\langle f_2,0.22\rangle$
TweetData	topic	$\langle 0,1,0\rangle$,[0-0.2) : $\langle f_4,0.08\rangle$, $\langle 0,1,0\rangle$,(0.2-0.4) : $\langle f_6,0.11\rangle$, $\langle 0,1,0\rangle$,(0.4-0.6) : $\langle f_4,0.18\rangle$, $\langle 0,1,0\rangle$,(0.6-0.8) : $\langle f_6,0.24\rangle$, $\langle 0,1,0\rangle$,(0.8-1] : $\langle f_6,0.26\rangle$

Table 7: `DecisionTable`.

tid	Topic BitMap	TopicOutput	Sentiment BitMap	Sentiment Output
t_1	[1,0,0]	[[0.18,0.64,0.05,...],[],[]]	[1,0,0]	[[0.94, 0.06,0], [], []]
t_2	[1,0,1]	[[0.5,0.2,0.1,...],[], [0.1,0.6,0.1,...]]	[1,0,1]	[[0.2,0.6,0.2],[], [0.86,0.1,0.04]]
t_3	[0,1,0]	[[], [0.78,0.06,0.02, ...], []]	[1,1,0]	[[0.1,0.7,0.2], [0.2,0.8,0],[]]

Table 8: `TweetDataState` table (created for tuples in `TweetData` table).

EnrichDB schema manager maintains metadata about EnrichDB relations, enrichment functions, as well as, function-families, and the decision table (will be clear soon). All the metadata is stored as tables (in PostgreSQL) and is maintained separately from the PostgreSQL system catalog (since we provide a layered implementation without changing the metadata store of underlying DBMS). EnrichDB catalog consists of the following components:

Metadata of EnrichDB relations. `RelationMetadata` stores names of each EnrichDB relation, names of attributes, its type — derived/fixed, the domain size in case of derived attributes, and a column `StateCutoff` — the purpose of this will be clear in §3.2. An example is shown in Table 4.

Metadata of enrichment functions. It is stored in `FunctionMetadata` table. EnrichDB supports three types of functions: (*i*) functions implemented in Python, using standard ML libraries, *e.g.*, scikit-learn [24] and TensorFlow [4], (*ii*) analytics library of Apache MADlib [13], where user can train ML algorithms using SQL, and (*iii*) C++/Java functions hosted using a REST API.

`FunctionMetadata` table stores: the column names of EnrichDB tables that contain the feature vector, derived attribute name, the parameters, the average execution time of function, and the quality of the function (see Table 6). Some of these functions require a training model. If a model is trained using EnrichDB `training procedure`, the model is stored in a `model_table`. For such functions, `FunctionMetadata` table stores `model_table` name. EnrichDB's `model_table` follows a similar format as in MADlib (`model_table` is discussed in Appendix 8.2).

Metadata of Function-families. It is stored in `FunctionFamily` table with columns: `FamilyID` , `FID` (function ID), `AttributeName`, and `RelationName`. Functions enriching different derived attribute, may belong to different function-families. An example is shown in Table 5.

Decision table. Since output of enrichment functions can be probabilistic, the set of functions that are used to enrich an attribute influences the uncertainty associated with the attribute. In ENRICHDB, a data owner/analyst can dictate which functions are used in which order to enrich data using a `DecisionTable`. It stores, for each derived attribute of a relation, a map that — given the current state of a tuple with respect to the attribute — specifies the next function that should be executed to further enrich the attribute, as well as (optionally) the measure of ***benefit*** that is expected to result.

While users can specify customized `DecisionTable`s that differ based on the mapping functions used to guide the enrichment process in ENRICHDB, by default, ENRICHDB uses a mapping function that — given a state of the attribute — determines the reduction in entropy [15] to guide which function should be executed next. That is, given the state, map specifies the function resulting in the maximum reduction of uncertainty per unit cost along with expected entropy reduction as a measure of benefit. For a probabilistic distribution over a set of domain values $D_1,...,D_n$ with probabilities $p_1,...,p_n$, entropy is measured as: $\sum_{i=1}^{n} -p_i \log p_i$.

Table 7 shows an example `DecisionTable`. Each row stores a map containing (state bitmap, entropy range) as keys and corresponding (next best function, benefit) pair as values. In first row the key is ($\langle 1,0,0 \rangle$,(0-0.25))) and value is ($\langle f_3, 0.2 \rangle$). To understand how this map is used, consider the state of tuple t_1 of `TweetData` (see Table 8) with sentiment state bitmap [1,0,1] and sentiment state output [[0.94,0.06,0],[0,0,0],[0,0,0]]. The entropy of t_1 is $(-0.94 \times \log(0.94) - 0.06 \times \log(0.06)) = 0.32$. From first row of Table 7, since entropy of t_1 is in the range (0.25-0.5), ENRICHDB determines the next best function to execute on t_1 is f_3 and its benefit as 0.2.

Learning of mapping function. ENRICHDB supports a mechanism to learn the default mapping function (*i.e.*, entropy-based) in `DecisionTable` for derived attributes. The mapping function uses the validation dataset (as introduced in §2). For each tuple in the validation dataset, ENRICHDB executes all permutations of available enrichment functions to capture the possible states of the attribute. Note that although we must capture all possible states, we do not have to run the enrichment functions multiple times. The output of a function, which is already executed on a tuple in the validation dataset, is stored in a temporary table, and only the combiner function is executed multiple times to simulate the state of attributes.

This execution results in an uncertainty value per tuple per state in the training dataset. We group tuples in the same state based on their uncertainty values, thereby each group corresponds to a single range (*e.g.*, $[0-0.25)$,$[0.25-0.5)$,..., see the first row of Table 7). Next, for each group, we find an enrichment function (from the remaining functions) that reduces the uncertainty value of the group tuples the most. Finally, we set the benefit of that group to the average uncertainty reduction, obtained from that function execution on all tuples of the group.

While ENRICHDB supports learning of the above map based on entropy, users can provide their own custom maps to guide the selection of functions that ENRICHDB uses for further enrichment. Alternatively, ENRICHDB supports a sampling-based method (not depending on the decision table) to execute enrichment, as discussed in §4.2.

3.2 State Manager

State manager stores the enrichment states of tuples of each relation in a `State` table (see Table 8) that is created at the time of creation of ENRICHDB tables. For each derived attribute $\mathcal{A}_i$, the `State` table stores the state-bitmap and state-output. *E.g.*, if there are three enrichment functions and the first function have been executed on tuple t_1 of `TweetData` table's `topic` derived attribute, then the `TopicStateBitmap` column stores $[1,0,0]$ and `TopicStateOutput` column stores the corresponding outputs.

ENRICHDB stores the state of tuples as a separate table instead of in-lining with ENRICHDB tables, since the size of the `State` table is often much smaller compared to ENRICHDB tables (that may contain tuples with binary large objects (BLOB), *e.g.*, images and tweets). By storing the `State` table separately, we significantly reduce the read/write cost of updating the state during query execution, as state update is a frequent operation during enrichment (will be clear in §4.2).

State Cutoff Representation. `StateOutput` column implemented naively can be large depending on the domain size of the derived attribute. *E.g.*, if the domain size of `topic` is 40 and there are 3 enrichment functions, then `TopicStateOutput` column (see Table 8) may contain 120 values in each row. Thus, large domain size may incur storage overhead and read/write cost of such states.

To address this problem, ENRICHDB uses a compressed representation to store `StateOutput` of tuples for attributes with large domain sizes. For such attributes, ENRICHDB sets up a ***cutoff threshold*** to reduce the storage size. The `StateOutput` value is stored as a set of key-value pairs, where the key is one of the domain values and the value is the probability of the tuple containing the domain value. Only the domain values with the probability higher than the cutoff threshold, are stored. This ensures that the tail-end of the distribution is not stored in `StateOutput` of tuples. In some cases, this strategy can require re-executions of enrichment functions, as probabilities corresponding to some domain values might be missing, *e.g.*, when a threshold based determinization function is used with a threshold less than the cutoff threshold. There is a tradeoff between the state size and the amount of re-executions of enrichment functions required during query execution. A higher cutoff threshold lowers the state size but increases the number of re-execution of enrichment functions and vice-versa.

Our strategy is motivated by the idea proposed in [16] in the context of *Uncertain Primary Indexing* (UPI) on uncertain data, where authors maintain a probability threshold (called *cutoff-threshold*) to store a subset of tuples of a relation in a faster primary index and the remaining tuples in a slower secondary index.

3.3 User Interfaces

A user interacts with EnrichDB using: (*i*) a *command-line-interface* (CLI) based tool and (*ii*) a *web-application*.

Command line interface (CLI). This interface allows a user to execute any arbitrary SQL query on EnrichDB using command line. The user needs to specify the query and two parameters of $epoch\_duration$ and max_epoch. The output of the query is maintained in a materialized view called `query-output`. After each epoch, a user needs to pose a query on the incremental materialized view (`SELECT * FROM query-output`) to retrieve the latest results or pose a query on the delta tables to. Apart from submitting a query, user can choose to `pause`, `resume` and `stop` an ongoing query from this interface.

EnrichDB web application. EnrichDB installation comes with a web-application for visualizing query results. Users can visualize both set based and aggregation based queries in this interface. The query is specified in the `query` field and then the user has to check the appropriate box (set-based or aggregation based query) to render appropriate visualization pages. In this interface, EnrichDB refreshes the screen when new results are available at the end of an epoch, new tuples are highlighted in green, the tuples which were the same as previous epoch are highlighted using blue and the deleted tuples are highlighted using red. An example query execution using the web application is shown in Figure 2. For an aggregation query, the application shows the graphs in a single page. The type of graphs depend on the type of the aggregation query: *i.e.*, query with `GROUP BY` clause and query without `GROUP BY` clause. For aggregation queries, the interface is similar to the interface of online aggregation [14]. The only difference is that a `GROUP BY` key can be added or removed in a later epoch.

4. QUERY PROCESSING

Query processing in EnrichDB consists of four main steps (Figure 3): query setup (§4.1), planning (§4.2), execution (§4.3), and interface update (§4.4). Query setup step is executed only once in the first epoch, and all remaining steps are executed iteratively once in each epoch. Note the **first epoch is a special epoch** in which EnrichDB sets up tables used during the enrichment plan generation phase (will be clear in §4.2). An EnrichDB query is wrapped in a stored procedure, called EnrichDB *executor* that internally executes the above steps.

Query setup involves rewriting the query to execute it on the determinized representation (see §2) of relations and produce the initial results, depending on tuples already enriched during ingestion/prior query executions. The planning step produces an enrichment plan for the next epoch based on the current state of tuples. The execution step performs enrichment on data tuples based on the generated plan. At the end of an epoch, the interface update step computes *delta answers* (the difference between the previously produced answers and the current answers, *i.e.*, $\Delta(q,e_k)$ and $\nabla(q,e_k)$ as specified in Equation 1) based on the new enrichment performed and updates the previously returned query results.

To execute these steps, EnrichDB includes the following UDF to support determinization of tuples (as described in §2).

Determinization UDF ($\boldsymbol{DET()}$): takes the state of a tuple as input and outputs the determinized representation of the tuple, *i.e.*, a tag, for a derived attribute. $DET()$ UDF, internally, calls a ***combiner function*** that combines the output of multiple enrichment functions of a derived attribute as described in §2.

4.1 Query Setup

Query setup performs the following tasks: (*i*) query rewrite to deal with determinized representation and (*ii*) computation of the initial set of results, without further enrichment. We explain the query rewrite process using a sample query shown in Figure 4(a) that involves three relations $R(\mathcal{A}_1, A_5)$, $S(\mathcal{A}_2, A_3)$, $T(A_4, A_5)$, where $\mathcal{A}_1$, $\mathcal{A}_2$ are derived and A_3, A_4, A_5 are fixed attributes. The rewritten query is shown in Figure 4. The query rewriting process contains the following stages:

1. In order to find relations that require determinization, EnrichDB lists all relations that have query predicates/projection on derived attributes (*e.g.*, R and S in our example Figure 4).
2. To find the determinized representation of tuples, the relations are joined with their corresponding `State` tables (*e.g.*, R is joined with R`State`) and then passed to $DET()$ UDF.
3. For a join involving derived attributes, join condition is replaced by a condition on the output of $DET()$ UDFs.

Note that the join condition on a table that does not require finding determinized representation, can be evaluated by underlying DBMS itself (*e.g.*, table T in Figure 4), and do not need determinization.

Example 4.1. We show the query rewrite strategy of EnrichDB for the query shown in Query 1 on `TweetData` table. The following SQL query is the rewritten query of EnrichDB *directly* executed on PostgreSQL. PostgreSQL can select its underlying optimization criteria that, however, does not affect query semantics.

```
CREATE
    INCREMENTAL MATERIALIZED VIEW mat_view AS
SELECT Tweet,location,TweetTime,topic,
```

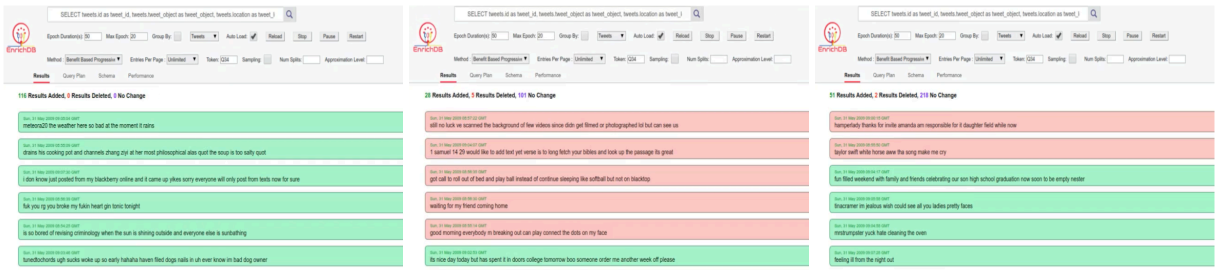

Figure 2: ENRICHDB web interface. The images show the query results at the end of three epochs during query execution. Green box shows newly added tuples and red box shows deleted tuples from previous epoch.

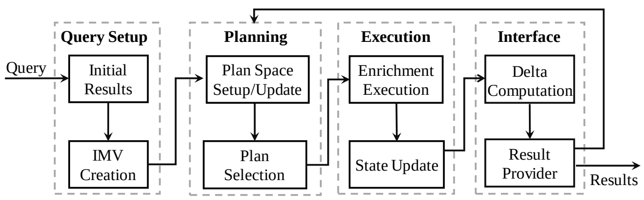

Figure 3: Query processing stages in *executor* procedure.

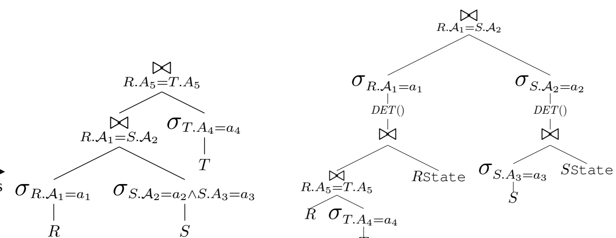

Figure 4: (a) Original Query (lhs) (b) Rewritten Query (rhs).

```
sentiment FROM (SELECT * FROM TweetData
WHERE
      TweetTime BETWEEN ('16:00','18:00')) as R
NATURAL JOIN TweetDataState AS RState WHERE
sentiment='positive' AND topic='social media'
```

Rewrite Optimizations. The layered implementation and query rewrite cannot fully ensure that the placement of operators and join order selected by PostgreSQL optimizer is optimal for ENRICHDB. Thus, below, we describe several optimizations performed by ENRICHDB during the query rewrite step. ENRICHDB adds query hints in the rewritten query, to help PostgreSQL's optimizer to select these query plans. These optimizations reduce the number of tuples that need to be considered for enrichment (thus reducing plan generation time), and reduce the number of tuples that need to be determinized.

• **Selection condition on fixed attributes.** ENRICHDB pushes down the selection condition on fixed attributes in the query tree. Consider a query shown in Figure 4(a), where the selection conditions on a relation involve both fixed and derived attributes and connected using AND operator (*i.e.*, $S.\mathcal{A}_2 = a_2 \wedge S.A_3 = a_3$). The rewritten query of ENRICHDB is shown in Figure 4(b). For more complex selection conditions, ENRICHDB converts the expression into CNF form and pushes all the conjuncts that involve only fixed predicates down the query tree to reduce the number of tuples that would require enrichment.

• **Join condition on fixed attributes.** If a query involves a join over only fixed attributes of tables, then ENRICHDB performs such a join before performing the determinization of the tables involved in the query (*i.e.*, prior to join of R and R`State` tables). Consider the query of Figure 4(a), where join condition (*i.e.*, $R.A_5 = T.A_5$) is on a fixed attribute. In the rewritten query (see Figure 4(b)), ENRICHDB joins R with T before joining it with R`State` and filters out the tuples of R that do not match the condition of $R.A_5 = T.A_5$. This optimization is useful for certain kind of joins, *e.g.*, foreign-key joins or when fewer tuples of R joins with tuples of T (*i.e.*, joins with low cardinality), as it reduces the number of tuples in R to be enriched.

Incremental Materialized View (IMV). Since ENRICHDB performs data enrichment in epochs, the state of tuples changes across different epochs; thus, query results also change across epochs. To capture *delta answers* across epochs, ENRICHDB creates an incremental materialized view (IMV) [7, 21, 17, 2] on the rewritten query (as shown in Example 4.1). Therefore, after enriching tuples in each epoch, their state change triggers an automatic update of IMV. Traditionally, materialized views are updated periodically using a command (*e.g.*, `REFRESH` command of PostgreSQL), which is inefficient due to evaluation of the query from scratch to rebuild the materialized view. Instead, IMV uses a relational algebra expression to compute and update delta changes of the view (using triggers, marked tuples, and temporary tables), which is more efficient. In the first epoch, the initial query results — produced by the rewritten query using the current state of tuples without any further enrichment — are stored in IMV, and in later epochs, delta answers are inserted/updated/deleted from the IMV.

4.2 Enrichment Plan Generation

An *enrichment plan* for a query in an epoch (say e_κ) consists of a set of ⟨tuple, enrichment function⟩ pairs, where the enrichment function will be executed on the tuple in e_κ. These plans are selected at the beginning of each epoch using either benefit- or sampling-based plan generation

Algorithm 1: EnrichDB plan generation.

1 **Inputs:** $\langle q,\delta,\mathtt{state}(R_i),type,\mathrm{DT},\kappa\rangle$. // q: a query on relations $R_1,...,R_n$, δ: epoch duration, *type*: type of plan generation - sampling or benefit, DT: `DecisionTable`, and κ: epoch id.
2 **Outputs:** Enrichment plan, *i.e.*, `PlanTable`, for q for epoch κ.
3 **Variables:** $\boldsymbol{f_m}$: enrichment function. ***benefit***: benefit of enrichment.
5 **Function** $\boldsymbol{GeneratePlan(\kappa)}$ **begin**
7   **if** $\kappa = 1$ **then** `PlanSpaceTable` $\leftarrow$ $PlanSpaceSetup(type)$;
9   **else** `PlanSpaceTable` $\leftarrow PlanSpaceUpdate(type)$;
11   `PlanTable` $\leftarrow$ $SelectBestFeasiblePlan(type,$`PlanSpaceTable`$,\delta)$
13   Return `PlanTable`
15 **Function** $\boldsymbol{PlanSpaceSetup(type)}$ **begin**
17   **for** $\forall R_i \in q$ **do**
19     $attribute\_list[] \leftarrow GetDerivedAttributeQuery(q(R_i))$
21   **for** $\forall(j,\ell) \in [\mathcal{A}_j \in attribute\_list[]] \times [t_\ell \in R_i]$ **do**
23   **if** $type =$ benefit **then**
25         $f_m, benefit \leftarrow GetNextFunction(\mathtt{state}(t_\ell.\mathcal{A}_j),\mathrm{DT})$
27   Append $\langle Name(R_i),ID(t_\ell),Name(\mathcal{A}_j),f_m,benefit\rangle$ to `PlanSpaceTable`
28   **else**
30         $f_m \leftarrow GetNextFunction(\mathtt{state}(t_\ell.\mathcal{A}_j))$
32   Append $\langle Name(R_i),ID(t_\ell),Name(\mathcal{A}_j),f_m\rangle$ to `PlanSpaceTable`
34   Return `PlanSpaceTable`
36 **Function** $\boldsymbol{SelectBestFeasiblePlan(type,}$**`PlanSpaceTable`**$\boldsymbol{,\delta)}$ **begin**
38   **if** $type =$ benefit **then**
40   `PlanTable`$\leftarrow$ $GetTopTuples(Sort($`PlanSpaceTable`$,benefit),\delta)$
42   **else** `PlanTable`$\leftarrow$ $GetRANDSample($`PlanSpaceTable`$,\delta)$;
44   Return `PlanTable`
46 **Function** $\boldsymbol{PlanSpaceUpdate(type)}$ **begin**
48   **for** $\forall\langle Name(R_i),ID(t_\ell),Name(\mathcal{A}_j),f_m,benefit\rangle \in$ `PlanTable` **do**
50   **if** $type =$ benefit **then** $f_p, benefit \leftarrow GetNextFunction(\mathtt{state}(t_\ell.\mathcal{A}_j),\mathrm{DT})$;
52   **else** $f_p \leftarrow GetNextFunction(\mathtt{state}(t_\ell.\mathcal{A}_j))$;
54   **if** $f_p = \emptyset$ **then** Delete $\langle Name(R_i),ID(t_\ell),Name(\mathcal{A}_j),f_m,benefit\rangle$ from `PlanSpaceTable` ;
56   **else** Update the function for $Name(R_i),ID(t_\ell),Name(\mathcal{A}_j)$ to f_p in `PlanSpaceTable`. ;
58   Return `PlanSpaceTable`

method. The ***benefit-based*** (**BB**) method selects pairs of tuples and enrichment functions based on a specific `DecisionTable` (learnt by EnrichDB or provided by user as discussed in §3.1), while the ***sampling-based*** (**SB**) method *randomly* selects $\langle$ tuple, enrichment function $\rangle$ pairs. The effectiveness of a plan generation strategy is measured using the progressive score defined in §2.2.3. Importantly, in EnrichDB, users can incorporate their own plan generation methods by implementing functions $GetNextFunction()$ and $SelectBestFeasiblePlan()$, shown in Algorithm 1.

EnrichDB maintains a `PlanSpaceTable` (see Table 9 as an example) to guide the plan generation for a query q. Rows in `PlanSpaceTable` correspond to all relation names ($Name(R_i)$) included in q, their tuple identifiers ($ID(t_\ell)$), and derived attributes' name ($Name(\mathcal{A}_j)$) with $\langle f_m, benefit\rangle$, where f_m is the next function that should be executed on the tuple t_ℓ and its *benefit* (used only in BB method). Note that the number of rows in `PlanSpaceTable` can be large, later we discuss how to limit that. `PlanSpaceTable` represents a subset of all possible enrichment plans for the query q. A subset of entries in `PlanSpaceTable` is selected for execution during a given epoch, and we refer to this subset as `PlanTable` for the epoch. The cost of the selected plan is the summation of the cost of enrichment functions mentioned in rows of `PlanTable`. Note that for the plan to be valid (*i.e.*, executable during the epoch), the cost of the selected plan *must* be smaller than the epoch duration (δ). The selection of rows in `PlanTable` from `PlanSpaceTable` is constrained to ensure the validity of the selected plan.

Algorithm 1 shows the pseudocode of enrichment plan generation for an epoch e_κ, consisting of the following three phases:

Plan Space Setup (Lines 15-34): This step is executed *only in the first epoch* to create and initialize `PlanSpaceTable`. Given q, an entry (*i.e.*, $\langle$tuple ID, next best function to be executed$\rangle$) is added to `PlanSpaceTable` for each derived attribute and tuple of relation R_i used in q. The next best function is computed using $GetNextFunction()$. In the BB method, $GetNextFunction()$ selects a function (not yet executed) for each tuple, using the appropriate map stored in `DecisionTable` and the current state of the tuple (Line 25). In SB method, $GetNextFunction()$ randomly selects a function that is not yet executed on the tuple (Line 30).

Plan Selection (Lines 36-44). Now, from `PlanSpaceTable`, our aim is to create `PlanTable`. Thus, for an epoch, we select a set of tuples (*i.e.*, plan) from `PlanSpaceTable` based on BB or SB method (using $SelectBestFeasiblePlan()$) and store them in `PlanTable`.

In BB method, our goal is to pick a subset of tuples, that maximizes the total benefit and the total cost is bounded by δ. This problem can be seen as a bounded weight resource maximization knapsack problem [8], which is NP-hard. Thus, we use Weighted Shortest Processing Time first (WSPT) heuristic-based approach [8] to select a set of tuples from `PlanSpaceTable`. Using WSPT, EnrichDB, first, sorts `PlanSpaceTable` in decreasing order of benefit, selects top-η tuples from the sorted table, such that the cost of enriching η tuples is $\approx \delta$, and finally, places them in `PlanTable`.

The SB method selects a plan by selecting a random

Relation	TID	Attribute	FID	benefit*
R_1	1	$\mathcal{A}_1$	f_2	0.8
...				
R_1	100	$\mathcal{A}_4$	f_2	0.67
R_2	1	$\mathcal{A}_1$	f_5	0.58
...				
R_2	200	$\mathcal{A}_2$	f_6	0.72

Table 9: `PlanSpaceTable`.

Relation	TID	Attribute	FID
R_1	2	$\mathcal{A}_2$	f_3
R_1	3	$\mathcal{A}_4$	f_5
R_2	1	$\mathcal{A}_4$	f_2
R_2	2	$\mathcal{A}_2$	f_6

Table 10: `PlanTable`.

sample of tuples from `PlanSpaceTable`. While any sampling can be used (*e.g.*, uniform random, stratified random, or variational sub-sampling as used in [5, 23]), EnrichDB uses uniform random sampling.

Plan Space Update (Lines 46-58). Since the execution of a plan in an epoch alters the state of the data, EnrichDB needs to recompute `PlanSpaceTable` in the next epoch. However, `PlanSpaceTable` must not be computed completely from scratch in each epoch, due to its overhead. Thus, EnrichDB updates only those tuples of `PlanSpaceTable` that were part of `PlanTable` in the previous epoch, as for all tuples in `PlanTable`, a new next function f_p and the corresponding benefit value is computed (Line 50) and updated in `PlanSpaceTable`. Also, if the tuple is fully enriched (*i.e.*, all enrichment functions are executed), the tuple is removed from `PlanSpaceTable` (Line 54).

Pruning of `PlanSpaceTable`. The size of `PlanSpaceTable` and plan generation time is substantially reduced by using the query rewrite optimizations (given in §4.1). Tuples to be considered for enrichment are those that will, eventually, be passed through $DET()$ UDF in the query tree. Thus, tuples not satisfying query predicates on fixed attributes are not added to `PlanSpaceTable`. EnrichDB analyzes q to determine for each $R_i \in q$, a query corresponding to $Prune(q,R_i)$ that represents a maximal subset of R_i such that tuples in $R_i - Prune(R_i)$ do not affect the answer of q and thus do not need to enriched. Although, R_i itself is such a maximal subset, but to reduce the amount of enrichment during query processing, EnrichDB attempts to find the smallest such subset. To compute $Prune(q,R_i)$, EnrichDB converts the selection condition of q on R_i into a CNF form, and then, for each conjunct, restricts the condition to only that on fixed attributes (*i.e.*, replaces conditions on derived attribute by true). Such a transformation represents a generalization of the selection query condition on R_i that can be used to determine a superset of tuples in R_i that might need enrichment. If q contains a join condition between fixed attribute of R_i with another relation's fixed attribute, then $Prune(q, R_i)$ can further exploit a semi-join query with the join condition on the corresponding fixed attributes. Considering the query of Figure 4, the prune queries for each relations involved are as follows:

```
Prune(q,R):SELECT R.id FROM R,T WHERE R.A5=T.
    A5 AND T.A4=a4
Prune(q,S): SELECT S.id FROM S WHERE S.A3=a3
```

4.3 Enrichment Plan Execution

Enrichment plans obtained from Algorithm 1 (mentioned in §4.2) are executed using a UDF, called ***driver UDF***. The driver UDF can execute any enrichment function supported by EnrichDB. The driver UDF takes as input: (*i*) a tuple t_ℓ of relation R_i, (*ii*) enrichment function f_m, and (*iii*) arguments to f_m (*e.g.*, model table name). The driver UDF executes f_m on tuple t_ℓ and updates `StateBitMap` and `StateOutput` of t_ℓ in R_i`State` table.

All tuples of a table R_i that are part of `PlanTable` need to be enriched during the plan execution phase. To do so, EnrichDB generates a query called *enrichment query* that joins R_i with `PlanTable` (to fetch ⟨tuple, function⟩ pairs) and `FunctionMetadata` table (to fetch the arguments of enrichment functions). The joined output is passed through the driver UDF that enriches the tuples of R_i and updates their state. The following enrichment query enriches tuples of `TweetData` table based on `PlanTable`.

```
SELECT driver(TD.*,FM.*) FROM
  TweetData
     TD, PlanTable PT, FunctionMetadata FM
WHERE PT.RelName='TweetData' AND TD.TID=PT.TID
  AND FM.FID=PT.FID
```

While passing the above query to PostgreSQL, we expand '*' with all the column names of `TweetData` and `FunctionMetadata`.

4.4 Query Answer Generation

As seen in the previous section, states are updated for tuples that are part of a plan. Since the state tables are part of IMV definition (§4.1), the update of state tables triggers an (automatic) update of IMV. *E.g.*, if the state of x tuples changes from the previous epoch in `TweetDataState` table, it might result in a change to determinized representations of these x tuples. Thus, some of these tuples that were previously filtered, may be passed by $comp()$ UDF in the query tree. Similarly, some tuples that were earlier passed by $comp()$ UDF, may be filtered out due to the new determinized representation. Note that IMV supports incremental updates of any monotonic SPJ and aggregation query.

Users can fetch complete answers at the end of an epoch by querying the IMV. If the complete answer set is large, EnrichDB allows users to retrieve delta changes of the answers, *i.e.*, inserted/deleted/updated tuples from the previous epoch. EnrichDB provides the delta answers to users through the delta tables that are maintained by IMV. Delta

Relation	Predicate	Action	Stage	Priority
`TweetData`	`TweetText` LIKE '%covid%'	*mlp_classifier*()	Ingestion	1
`TweetData`	`TweetTime` >11:00 AND `TweetTime` <14:00	*dt_classifier*()	Background	1
`TweetData`	`location`='CA' AND `TweetText` LIKE '%president%'	*mlp_classifier*(), *dt_classifier*()	Ingestion	2

Table 11: Enrichment policies.

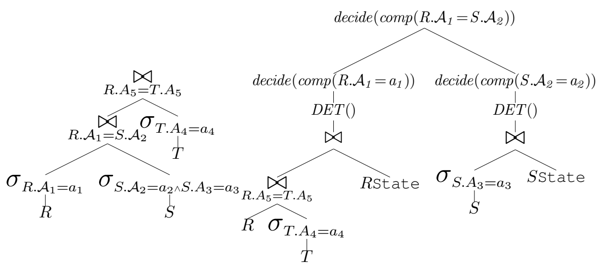

Figure 5: (a) Original Query(lhs) (b) Rewritten Query(rhs).

tables store the delta results that are computed from a delta relational algebra expression [21, 3] and they are used to update the materialized view incrementally by IMV. The delta answers can also be fetched by creating triggers on top of IMV. The current implementation allows users to fetch delta answers only from the last epoch. Fetching delta answers from any arbitrary epoch using a cursor is complex (will be supported in future version), since the query processing in EnrichDB is not demand-driven, as in SQL databases.

5. USE CASE FOR THE SYSTEM

In the case of **social media analysis**, let us consider an application, where analysts are interested to find the reaction of the public about the most recent presidential debate from tweets. To do so, analysts store the tweets after the end of the presidential debate in a table `TweetData`. Here, analysts must train multiple machine learning models for detecting the sentiment and topic of such tweets based on a past dataset (stored in `TweetsTrain` table) that was collected during the previous presidential debates. Finally, analysts pose a query on `TweetData` table to find out all tweets with positive sentiment and the topic of the presidential debate. In order to achieve these functionalities, the steps that analysts must take in EnrichDB are presented below. Note that EnrichDB-based implementation is much simpler (≈12 lines of code) as compared to any loosely coupled implementation, where enrichment is performed outside of DBMS and requires much more lines of code (≈130 lines, as shown in Appendix §8).

```
 -- Creating a new table
CREATE TABLE TweetData(tid char(8),
   userid char(20),Tweet text,feature float[],
   topic int derived:40,sentiment int
   derived:3, Time timestamp,location text);
-- Training ML Models
SELECT db.model_train('TweetsTrain',
   'sentiment_dt', 'decision_tree',
   'sentiment', 'feature[]', model_params);
-- Associating functions with 'sentiment'
SELECT db.assign_enrichment_functions(
   'tweets',
   [['sentiment',3,'sentiment_dt',0.8,0.7],
   ['sentiment',4,'sentiment_fo',0.9,0.8],
   ['sentiment',1,'sentiment_mlp',0.9, 0.9]]);
-- Setting up decision table
SELECT db.learn_decision_table('TweetData',
   'sentiment','TweetValidationSet');
--Adding data
SELECT db.enriched_insert
   ('INSERT INTO TweetData (id, tweet_object,
   topic,sentiment,timestamp,location) VALUES
   (1,''is very happy for my friend'',NULL,
   NULL,"2021-02-06 12:29:06","L0")');
-- Executing Queries
CALL db.exec_driver_udf('SELECT id,
   tweet_object,location,timestamp, sentiment,
   topic FROM TweetData WHERE sentiment = 0
   AND topic = 2 AND id < 10000', 20, 0.9);
```

6. RELATED SYSTEMS

EnrichDB can be viewed as a system similar to *Extract-Load-Transform (ELT)* based systems [1], where the data is extracted and loaded to a data warehouse/lake system and enrichment is performed at the analysis time. However, in contrast to such ELT systems, EnrichDB provides a powerful data model to application developers that make application programming very easy. *Query-driven approaches* of data cleaning has been studied significantly [29, 12]. However, such works were restricted to only data cleaning algorithms of duplicate detection, duplication elimination, and entity resolution, whereas EnrichDB supports a general class of enrichment functions such as classification functions, clustering functions, and regression functions. *Systems for supporting ML using databases* (*e.g.*, Apache MADlib [13], RIOT [32]) are designed to learn ML models inside or on top of database systems; however, such systems do not support semantic abstraction of specifying enrichment functions and linking them to higher-level semantic observation generated by them as supported by EnrichDB.

7. CONCLUSION

In this paper, we proposed EnrichDB — a new system for supporting data enrichment inside a single data management system. The cornerstone of EnrichDB is a powerful *enrichment data model* that encapsulates enrichment as an operator inside a DBMS enabling it to co-optimize enrichment with query processing. Furthermore, EnrichDB provides semantic abstraction, transparency of enrichment, and progressive computation of queries to make application programming very simple for the developers.

8. APPENDIX

8.1 Discussion on Model Table

This section describes the model table of EnrichDB using an example model of multi-layered perceptron. The model table follows the similar format of Madlib. The contents of the table for the MLP classifier is shown in Table 12. This table stores the source table name (*i.e.*, source_table) on which the model was trained, the column names used as the independent variables in model training (independent_varname), the name and data type of the predicting column (dependent_varname), and model specific parameters (*i.e.*, tolerance, learning_rate_init, learning_rate_policy, momentum, n_iterations, n_tries, layer_sizes, activation, is_classification, classes, weights, grouping_column). For the model tables of other classifiers, please check out the format in the documentation of Apache Madlib[13].

Attributes	Values
source_table	TweetData
independent_varname	feature
dependent_varname	sentiment
dependent_vartype	integer
tolerance	0
learning_rate_init	0.003
learning_rate_policy	constant
momentum	0.9
n_iterations	500
n_tries	3
layer_sizes	5
activation	tanh
is_classification	t
classes	0,1,2
weights	1
grouping_col	NULL

Table 12: Model table for multi-layered perceptron classifier.

8.2 Loosely Coupled Approach Codebase

In this section, we present the codebase of twitter analysis application developed using the data-flow system of Spark.

```
import time
import re
import sys
import numpy as np
import pickle
import pyspark
from pyspark import SQLContext
from pyspark.sql.types import
     StructType, StructField, IntegerType
    , FloatType, StringType, ArrayType
from pyspark.sql.functions import udf
from pyspark.sql import Row
from pyspark.sql.functions import col
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark import SparkConf
import pyspark.sql.functions as F
sys.path.insert
    (0, '/home/ubuntu/functions/backend/load')
url = 'jdbc:postgresql://localhost
    :5432?user=postgres&password=postgres'
table = 'tweets'
conf = SparkConf()
conf.setMaster("local[*]")
conf.setAppName('pyspark')


properties = {
    'user': 'postgres',
    'password': 'postgres',
    'driver': 'org.postgresql.Driver',
    'spark
    .jars':'org.postgresql:postgresql:42.2.12'
}
sc = pyspark.SparkContext.getOrCreate()
spark = SparkSession
    .builder.appName("Python Spark SQL
    ").config("spark.jars","/home/ubuntu/java
    /postgresql-42.2.18.jar").getOrCreate()
df = spark.read.format
    ("jdbc").option("url", "jdbc:postgresql
    ://localhost:5432/test").option("dbtable
    ", "tweets").option("user", "postgres
    ").option("password", "postgres").load()


clf_dt =
     pickle.load(open('/home/ubuntu/tweet_clfs
    /tweet_dt_sentiment_calibrated.p', 'rb'))
clf_gnb
    = pickle.load(open('home/ubuntu/tweet_clfs
    /tweet_gnb_sentiment_calibrated.p', 'rb'))
clf_lda
    = pickle.load(open('home/ubuntu/tweet_clfs
    /tweet_lda_sentiment_calibrated.p', 'rb'))
clf_mlp
    = pickle.load(open('home/ubuntu/tweet_clfs
    /tweet_mlp_sentiment_calibrated.p', 'rb'))
def execute_mlp(rl):
    gProb = clf_mlp.predict_proba(rl)
    return gProb[0]
def execute_dt(rl):
    gProb = clf_dt.predict_proba(rl)
    return gProb[0]
def execute_gnb(rl):
    cProb = clf_gnb.predict_proba(rl)
    return cProb[0]
def execute_lda(rl):
    gProb = clf_lda.predict_proba(rl)
    return gProb[0]
def execute_svm(rl):
    gProb = clf_svm.predict_proba(rl)
    return gProb[0]


def generateCombinedProbability
    (functionBitmap, probability2dArr):
```

```
    output_arr
     = [0] * len(probability2dArr[0])
    num_possible_tag
     = len(probability2dArr[0])

    weights = [1,2, 3 ,6]

    for i in range(num_possible_tag):
        sum_val = 0.0
        count_val = 0
        for j in range(len(functionBitmap)):
            if functionBitmap[j] == 1:
                sum_val
     += weights[j]*probability2dArr[j][i]
                count_val += weights[j]
        if count_val > 0:
            output_arr
    [i] = 1.0 * (sum_val / count_val)
        else:
            output_arr[i] = 0

    return output_arr

def _response(input_features):
    fcname = 'tweet_sentiment_all'
    response = None
    proba = None

    if fcname == 'tweet_sentiment_all':
        bitmap = [1,1,1,1]
        prob2DArray =[]

        features =  input_features
        proba = execute_dt([features[:1000]])
        prob2DArray.append(proba)

        proba = execute_gnb([features[:1000]])
        prob2DArray.append(proba)

        proba = execute_lda([features[:1000]])
        prob2DArray.append(proba)

        proba = execute_mlp([features[:1000]])
        prob2DArray.append(proba)

        output = generateCombinedProbability
    (bitmap, prob2DArray)
        proba = output
        response = 0

    res = [round(v,6) for v in output]
    max_val = max(res)
    label = res.index(max_val)
    return label


if __name__ == "__main__":

    start_id = 50000
    end_id = 60000
    _select_sql =
     "(select t1.id,t1.tweet_object,t1.feature
    , t2.sentiment from tweets t1, tweets_full
     t2 where t1.id = t2.id and t1.id >" +
    str(start_id)+ " and t1.id <" + str(end_id
    ) +  " and t2.id >" + str(start_id) +" and
     t2.id  <" + str(end_id) + ") as my_table"
    df_select = spark.read.jdbc(
    url="jdbc:postgresql://localhost:5432/test
    ",table=_select_sql,properties=properties)
    df_select.show()
    truth_list = df_select.select('sentiment
    ').rdd.flatMap(lambda x: x).collect()
    all_f1 = []
    for i in range(len(truth_list)):
        if truth_list[i] == 1:
                truth+=1
    prev_recall =0
    max_time = 50
    ep_size = 10
    quality_requirement = 0.9

    for i in range(max_time):
        t1 = time.time()
        start_id = 0 + i * epoch_size
        end_id = start_id + epoch_size
        _select_sql =
     "(select t1.id,t1.tweet_object,t1.feature
    , t2.sentiment from tweets t1, tweets_full
     t2 where t1.id = t2.id and t1.id >" +
    str(start_id)+ " and t1.id <" + str(end_id
    ) +  " and t2.id >" + str(start_id) +" and
     t2.id  <" + str(end_id) + ") as my_table"
        df_select = spark.read.jdbc(
    url="jdbc:postgresql://localhost:5432/test
    ",table=_select_sql,properties=properties)
        df_select.show()
        output_udf_float
     = udf(_response, IntegerType())

        df4 = df_select
    .withColumn('exec',output_udf_float
    ('feature').alias('exec_output'))
        df4.show()
        query_res = df4.select(
    'exec').rdd.flatMap(lambda x: x).collect()

        count = 0
        correct = 0
        for j in range(len(pred_list)):
            if query_res[j] == 1:
                count+=1
            if query_res[j] == truth_list
    [j + i*ep_size] and pred_list[j] == 1:
                correct+=1

        prec = correct*1.0/count

        recall = correct* 1.0 /truth
        total_recall = prev_recall + recall
        prev_recall = total_recall
        if (prec + total_recall) != 0:
            f1 = 2* prec
     * total_recall / (prec + total_recall)
        else:
            f1 = 0
        all_f1.append(f1)

       if f1 >= quality_requirement:
        break
    query_res.show()
```

9. REFERENCES

[1] Apache airflow. https://airflow.apache.org/.

[2] Incremental view maintenance for postgresql. https://wiki.postgresql.org/wiki/Incremental_View_Maintenance.

[3] Incremental view maintenance implementation for postgresql. https://github.com/sraoss/pgsql-ivm.

[4] Tensorflow. https://www.tensorflow.org/.

[5] S. Agarwal et al. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42. ACM, 2013.
[6] Y. Altowim et al. Progressive approach to relational entity resolution. *VLDB 2014*.
[7] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.
[8] P. Brucker. *Scheduling Algorithms*. Springer Publishing Company, 2010.
[9] R. Cheng et al. Cleaning uncertain data with quality guarantees. *VLDB 2008*.
[10] W. Cheng et al. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, 2010.
[11] S. Feng et al. Uncertainty annotated databases - A lightweight approach for approximating certain answers. In *SIGMOD*, 2019.
[12] S. Giannakopoulou et al. Cleaning denial constraint violations through relaxation. In *SIGMOD*, 2020.
[13] J. M. Hellerstein et al. The madlib analytics library or MAD skills, the SQL. *VLDB 2012*.
[14] J. M. Hellerstein et al. Online aggregation. *SIGMOD Rec.*, 1997.
[15] A. Holub, P. Perona, and M. C. Burl. Entropy-based active learning for object recognition. In *CVPR Workshops*, pages 1–8. IEEE Computer Society, 2008.
[16] H. Kimura, S. Madden, and S. B. Zdonik. UPI: A primary index for uncertain databases. *Proc. VLDB Endow.*, 3(1):630–637, 2010.
[17] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.
[18] M. Lapin et al. Top-k multiclass SVM. In *NIPS 2015*.
[19] Y. Lin et al. LOCATER: cleaning wifi connectivity datasets for semantic localization. *VLDB 2020*.
[20] N. May et al. Sap hana–the evolution of an in-memory dbms from pure olap processing towards mixed workloads. *BTW 2017*.
[21] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264. ACM, 2014.
[22] T. Papenbrock et al. Progressive duplicate detection. *TKDE*, 2015.
[23] Y. Park et al. Verdictdb: Universalizing approximate query processing. In *SIGMOD Conference*, pages 1461–1476. ACM, 2018.
[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[25] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *Proc. VLDB Endow.*, 7(13):1722–1729, 2014.
[26] D. Suciu et al. Probabilistic databases. *Synthesis lectures on data management*, 3(2):1–180, 2011.
[27] J. A. K. Suykens et al. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, 1999.
[28] X. Wang et al. An IDEA: an ingestion framework for data enrichment in asterixdb. *Proc. VLDB Endow.*, 12(11):1485–1498, 2019.
[29] S. E. Whang et al. Pay-as-you-go entity resolution. *IEEE TKDE*, 2013.
[30] D. H. Wolpert. Stacked generalization. *Neural Networks*, 1992.
[31] M. Zaharia et al. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.
[32] Y. Zhang et al. I/o-efficient statistical computing with RIOT. In *ICDE 2010*.