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.*, sensor-driven smart spaces and social media analytics) 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. Data analysis pipeline refers 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 sensor interpretation and fusion over sensory inputs, mechanisms for sentiment analysis over social media posts, and named entity extraction in text.

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.

[14] has highlighted the limitations of the traditional data warehouse approach in analyzing the recent data (as it arrives) for 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 [20] often used for scalable ingestion) are capable of executing enrichment functions on newly arriving data prior to its storage in a DBMS. Recently, [17] 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 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.

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.

Motivating Example. A quintessential example domain for which ENRICHDB is designed, is a sensor-driven smart space environment. Such an environment is often instrumented with a large number of sensors producing data, which is stored in databases. Such data consists of videos, images, data from motion sensors, as well as connectivity data of user's mobile devices with WiFi access points. Such data needs to be processed before it can be used by applications. *E.g.*, [12] uses connectivity data of user's mobile devices with WiFi access points to localize users inside a building. Furthermore, one can use surveillance camera images to localize users more accurately. Localization based on WiFi connectivity data or images can be expensive, *e.g.*, analyzing a single WiFi

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

connectivity event takes ≈200ms, and analyzing a single image takes ≈1s. If we consider a campus environment with hundreds of WiFi access points and cameras (where ≈1,000 Wi-Fi events/sec and ≈100 images/sec are produced by the sensors), we will need ≈5 minutes of processing time for locating person using the data that has been generated in one second, and such a processing time is not feasible.

Instead, we need to process such data during query execution in an adaptive manner. Queries on such data can be ad-hoc: for example, a visitor planning to attend an event at a location may wish to know the attendees already arrived (or the count) apriori to avoid crowded regions. Another example will be exploring suspicious activities that may create a timeline of events at different parts of a building using WiFi connectivity data and then performing detailed analysis using camera images. To answer such ad-hoc queries, if a system enriches the required data at query time, it can still result in high latency depending on the query selectivity. ■

Motivated by the above 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 and Transparency of Enrichment. EnrichDB supports a declarative interface to specify and to 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 have to deal with raw data directly — applications can be fully developed based on higher-level semantic observation. Furthermore, developers do not have to be concerned about what data has to be enriched, using which functions, and at what stage of data processing. EnrichDB maintains the state of enrichment of objects and performs enrichment automatically based on the current state of objects.

Optimization of Enrichment. EnrichDB allows enrichment all through the data processing pipeline. EnrichDB makes sure that enrichment of objects is performed optimally. At query time enrichment, EnrichDB exploits query optimizer to prune away enrichment of objects that do not influence the query results. Furthermore, EnrichDB allows enrichment of data closer to where the data resides resulting in a low data movement.

Progressive Computation. When EnrichDB executes enrichment functions during query processing, it produces query answers progressively. A progressive query answering technique (motivated by Approximate Query Processing systems [10] that provided progressive query answering for aggregation queries) 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. The codebase and detailed discussion on design decisions are presented in [2].

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 wifi(id int, user_id char(30),
  timestamp time, wifi_ap char(30),
  location int derived:304);
```

The value of a derived attribute is determined using one or more ***enrichment functions*** associated with it.[2]

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 [16], (*ii*) *multi-valued*: outputting a set of values, *e.g.*, top-k classifiers [11], (*iii*) *probabilistic*: outputting a probability distribution over the possible values of a label, *e.g.*, probabilistic classifiers [6]. Also, enrichment functions can be categorized based on the size of output domain: (*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 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 that internally uses machine learning models to predict the value of derived attributes. Often such models use a supervised learning method [5] that learns a mapping function between a set of input and output pairs from a ground truth data set (often referred to as training data). A user needs to

[2]The derived attributes cannot be updated directly by the user.

id	user_id	time	wifi_ap	location
t_1	24	09:14	56	L1
t_2	22	10:26	110	NULL
t_3	108	14:10	116	L4

Table 1: The `wifi` table (`location` is derived).

specify the table that stores the training data for the model. Below, we show an example where a machine learning model of Multi-Layer Perceptron (MLP) is learned using a training procedure of `model_train`. The training data is stored in `wifi_train` table and the name of the model is `location_mlp`. It uses the attribute values of `feature` as input to the model and outputs the prediction for `location` attribute. The model-specific parameters are passed as a string in `model_params`.[3]

```
SELECT db.model_train('wifi_train',
   'location_mlp', 'mlp','location',
   'feature[]', model_params);
```

The *cost* and *quality* of enrichment functions can either be specified by user or can be determined automatically by using several methods, *e.g.*, train/test split and k-fold cross-validation during the training phase.

In real scenarios, often multiple enrichment functions are used to perform a particular analysis. To localize a person, one can use multiple ML functions, *e.g.*, decision tree, random forest, and multi-layered perceptron models. EnrichDB supports specification of such functions using a function-family. Formally, 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 [19] combiner functions. Below we show, creation of function-family for `location` attribute consisting of multiple functions along with their cost (seconds/tuple) and quality (measured in AUC) respectively, using the `assign_enrichment_functions` command.

```
SELECT db.assign_enrichment_functions('wifi',
    [['location',3,'location_dt',0.8,0.7],
   ['location',4,'location_fo',0.6,0.8],
   ['location',1,'location_mlp',0.95, 0.9]]);
```

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 three enrichment functions f_1, f_2, f_3 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

[3]If a machine learning model is updated, *i.e.*, re-trained, then a new enrichment function has to be added that uses the retrained model.

tid	location
t_1	L1:0.54, L2:0.35, L3: 0.11
t_2	L1: 0.1, L2: 0.1, ..., L10: 0.1
t_3	L4:0.8, L5: 0.15, L6: 0.05

Table 2: State output for derived attributes.

$t_k.\mathcal{A}_i$ contains $\langle 101\rangle$, *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. For single/multi-valued functions and continuous 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` (see Table 3).

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 expected improvement in quality (denoted as ***benefit***) that will result from enriching the attribute of the tuple. EnrichDB uses benefit and the cost of enrichment functions to order the enrichment of tuples.

In Table 3, each row stores a map containing (state bitmap, entropy range) as keys and the corresponding (next best function, benefit) pair as values. Consider the tuple t_1 of `wifi` table (see Table 1) and assume that the location state bitmap of t_1 is [1,0,0] and the location state output of t_1 is [[0.54,0.35,0.11], [0,0,0], [0,0,0]]. The entropy of t_1 is $(-0.54 \times \log_3(0.54) - 0.35 \times \log_3(0.35) - 0.11 \times \log_3(0.11)) = 0.85$. From first row of Table 3, since entropy of t_1 is in the range (0.75-1], the decision table specifies that the next best function to execute is f_2 and its benefit as 0.22.

2.2 Query Model

This section describes the query language (§2.2.1), query semantics (§2.2.2), and the goal of enrichment (§2.2.3).

2.2.1 *Query Language*

The query language of EnrichDB is an extended

Rel.	Attribute	Map
wifi	location	$\langle 1,0,0\rangle$, [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$\rangle$
wifi	location	$\langle 0,1,0\rangle$, [0-0.5) : $\langle f_4, 0.08\rangle$, $\langle$0,1,0$\rangle$,(0.5-1] : $\langle f_6$,0.11$\rangle$

Table 3: A part of `DecisionTable`.

version of SQL. Queries in ENRICHDB are associated with a 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 in the past: (*i*) determinization-based semantics [7] and (*ii*) possible world (PW) semantics [15]. The determinization-based semantics converts probabilistic representation to a single or a small set of deterministic worlds. The query is executed in each of 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 choice of one semantics over the other depends on the application scenarios. In some scenarios, applications can make good decisions by using the most probable answers, whereas in some scenarios, they 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 that requires a minimum quality of 0.9 is shown below:

```
SELECT wifi.location as p_location,
wifi.timestamp as p_time  FROM wifi
WHERE p_location = 'L1'
AND p_time BETWEEN ('10:00','12:00')
AND QUALITY 0.9;
```

2.2.2 *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:

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

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 4: Truth table for evaluating complex conditions.

In Table 1, `location` attribute stores the determinized value (using top-1 determinization strategy) based on the state stored in Table 2.

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 ***T***, ***F***, and ***U*** into a four-valued logic: **true (*T*)**, **false (*F*)**, **possible (*P*)**, and **unknown (*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 4 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 lower bound l, while applied to all T and P tuples produces the upper bound u. *E.g.*, consider a query on Table 1 that counts the occupancy of location L1, 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 then ranks their outputs by 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 locations with highest occupancy from Table 1. Suppose after applying $count()$, the locations had following bounds for occupancy: L1: [100,150], L2:[110,120], L3:[100,115], and L4:[80,95]. The results returned are locations {L1, L2, L3} that guarantees that the actual top-2 locations (*i.e.*, L1, L2) are part of the result. L4 is excluded as the upper bound of occupancy (*i.e.*, 95) is lower than the lower bounds of locations in the answer.

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 ENRICHDB, 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.

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 [13, 4]):

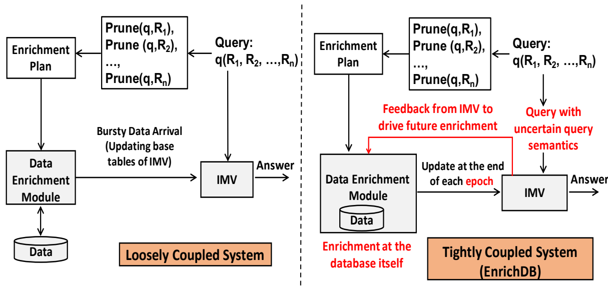
Enrichment Plan
Prune(q,R$_1$), Prune (q,R$_2$), ..., Prune(q,R$_n$)
Query: q(R$_1$, R$_2$, ...,R$_n$)
Bursty Data Arrival (Updating base tables of IMV)
Data Enrichment Module
IMV
Answer
Data
Loosely Coupled System
Enrichment Plan
Prune(q,R$_1$), Prune (q,R$_2$), ..., Prune(q,R$_n$)
Query: q(R$_1$, R$_2$, ...,R$_n$)
Feedback from IMV to drive future enrichment
Query with uncertain query semantics
Update at the end of each epoch
Data Enrichment Module
Data
IMV
Answer
Enrichment at the database itself
Tightly Coupled System (EnrichDB)

Figure 1: Loosely coupled system versus ENRICHDB.

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

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 metric, and a set of weights assigned to each epoch, 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 shown in Figure 1: (*i*) a *loosely coupled* (LC) approach, wherein an enrichment module is implemented separately from the DBMS, and (*ii*) a *tightly coupled* (TC) approach, wherein an enrichment module is tightly integrated with the query processing module of the DBMS. ENRICHDB follows TC approach on top of PostgreSQL as it uses the query context to eliminate redundant enrichment. Consider a query with two selection conditions on derived attributes $\mathcal{A}_1$ and $\mathcal{A}_2$, connected using AND, the LC approach will enrich the tuples for both $\mathcal{A}_1$ and $\mathcal{A}_2$. In contrast, in TC, after enriching $\mathcal{A}_1$ of a tuple, if it does not satisfy the condition on $\mathcal{A}_1$, then attribute $\mathcal{A}_2$ is not enriched. Such a pruning strategy can be very effective, when queries are complex and selective. Furthermore, the TC approach executes the enrichment functions closer to the data, in the database engine.

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 functions as UDFs by taking them as arguments. For implementation details, please check [2].

4. USE CASE OF ENRICHDB

This section describes how ENRICHDB can be used to develop the application described in §1 that finds out location of attendees already arrived for an event. It requires fine-grained localization of people using WiFi connectivity data inside a building using multiple predictive models with different cost and quality [12]. The application poses queries to find out attendees at a location between two time intervals.
Ease of Application Development. To develop this application, the steps to take in ENRICHDB are presented below. ENRICHDB-based implementation is much simpler ($\approx$26 lines of code) as compared to any loosely coupled implementation, where enrichment is performed outside of DBMS and requires much more lines of code ($\approx$130 lines [2]).

```
1  -- Creating a new table
2 CREATE TABLE wifi(id int, user_id char(30),
3   time timestamp, wifi_ap char(30),
4   location int derived:304)
5 -- Training ML Models
6 SELECT db.model_train('wifi_train',
7   'location_dt', 'decision_tree',
8   'location', 'feature[]', model_params);
9 -- Associating functions with `location'
10 SELECT db.assign_enrichment_functions(
11   'wifi', [['location',3,'loc_dt',0.8,0.7],
12   ['location',4,'loc_fo',0.9,0.8]]);
13 -- Setting up decision table
14 SELECT db.learn_decision_table('wifi',
15   'location','WifiValidation');
16 --Adding data
17 SELECT db.enriched_insert('INSERT INTO wifi
18    VALUES (1,1051,"10:02",12, NULL)');
19 -- Executing Queries
20 call db.exec_driver('SELECT location , time
21    FROM wifi WHERE id<100 AND location ='L1'
22    AND time BETWEEN ("10:00","12:00")',20,5);
```

Performance Evaluation. Figure 2 shows the quality of results achieved by ENRICHDB with respect to time for the query described above (Line 20). The results are produced at the end of each epoch, where the epoch duration is set to 5 seconds. The quality is measured using ***normalized*** F_1 ***measure***, *i.e.*, F_1/F_1^{max}, where F_1^{max} is the maximum F_1 measure achieved during query execution. Figure 2 highlights that ENRICHDB provides high-quality query results within the first few epochs of a query execution as compared to the strategy of eager enrichment that enriches the tuples completely and then executes the query.

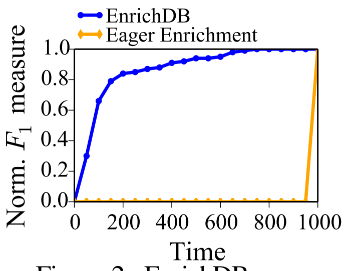

Figure 2: EnrichDB vs complete enrichment at query time.

5. 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. In contrast, ENRICHDB provides a powerful data model to developers that make application programming very easy. *Query-driven approaches* of data cleaning has been studied significantly [18, 8]. 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, clustering, and regression functions. *Systems for supporting ML using databases* (*e.g.*, Apache MADlib [9], RIOT [21]) 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 observation generated by them as supported by ENRICHDB.

6. 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.

Acknowledgements. This material was partially funded by the research sponsored by DARPA under agreement number FA8750-16-2-0021 and NSF Grants No. 1527536, 1545071, 2032525, and 2008993.

7. REFERENCES

[1] Apache airflow. `https://airflow.apache.org/`.
[2] Full paper and code. `https://github.com/DB-repo/enrichdb`.
[3] IMV implementation of postgresql. `github.com/sraoss/pgsql-ivm`.
[4] Y. Altowim et al. Progressive approach to relational entity resolution. *VLDB 2014*.
[5] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. ICML '06.
[6] W. Cheng et al. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, 2010.
[7] S. Feng et al. Uncertainty annotated databases - A lightweight approach for approximating certain answers. In *SIGMOD*, 2019.
[8] S. Giannakopoulou et al. Cleaning denial constraint violations through relaxation. In *SIGMOD*, 2020.
[9] J. M. Hellerstein et al. The madlib analytics library or MAD skills, the SQL. *VLDB 2012*.
[10] J. M. Hellerstein et al. Online aggregation. *SIGMOD Rec.*, 1997.
[11] M. Lapin et al. Top-k multiclass SVM. In *NIPS 2015*.
[12] Y. Lin et al. LOCATER: cleaning wifi connectivity datasets for semantic localization. *VLDB 2014*.
[13] T. Papenbrock et al. Progressive duplicate detection. *TKDE*, 2015.
[14] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *Proc. VLDB Endow.*, 7(13):1722–1729, 2014.
[15] D. Suciu et al. Probabilistic databases. *Synthesis lectures on data management*, 2011.
[16] J. A. K. Suykens et al. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, 1999.
[17] X. Wang et al. An IDEA: an ingestion framework for data enrichment in asterixdb. *Proc. VLDB Endow.*, 12(11):1485–1498, 2019.
[18] S. E. Whang et al. Pay-as-you-go entity resolution. *IEEE TKDE*, 2013.
[19] D. H. Wolpert. Stacked generalization. *Neural Networks*, 1992.
[20] M. Zaharia et al. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.
[21] Y. Zhang et al. I/o-efficient statistical computing with RIOT. In *ICDE 2010*.