EnrichDB: A Database System for Progressive Data Enrichment and Query Evaluation

Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, and Shantanu Sharma
University of California, Irvine, USA.

ABSTRACT

This paper proposes EnrichDB, a database system designed to support domains that require data to be enriched prior to use. Unlike several recent systems that have explored ways to optimize enrichment at ingestion, EnrichDB integrates enrichment all through the data processing pipeline — at ingestion, triggered based on events, and progressively during query processing. EnrichDB is designed for situations when enrichment is too expensive to be performed completely prior to usage. EnrichDB is implemented using a layered approach on top of PostgreSQL, though it can be easily layered on other databases. This paper describes the architecture, design choices, and implementation of EnrichDB, as well as, evaluates its performance focusing on EnrichDB's support for enrichment during progressive query answering.

1 INTRODUCTION

Today's databases require support for enriching raw uninterpreted data (*e.g.*, text, images, and sensor data) into semantically meaningful data that can be used for analytical applications. Examples of such enrichment include entity linking [27], sentiment analysis of tweets [17, 31], semantic interpretation of sensor data [69], and data transformations using machine learning (ML) algorithms to interpret/classify images [44, 65]. Such enrichment is usually expensive and cannot be fully performed on the data collected, specially when resources are limited.

Traditionally, data enrichment is performed as part of an offline Extract-Transform-Load (ETL) process that causes significant latency between the time data is generated and the time it is available for analysis [3, 20, 24]. Also, such enrichment is wasteful, when queries can be answered by enriching only parts of the data or enriching data partially using a subset of available enrichment techniques. Recently, several industrial systems [2, 24, 85, 93] have explored data enrichment at scale during ingestion, including techniques [86] to batch input data to reduce ingestion overhead. While such systems can be used when simple functions (*e.g.*, that do not add significant latency) are executed at ingestion, they do not scale to situations when data analysis requires execution of a suite of computationally-expensive functions. We illustrate the challenge of executing expensive enrichment functions on large datasets based our experiences in developing analytic applications that have served as a motivation behind developing EnrichDB.

Motivating Example. Consider an instrumented building, with hundreds of WiFi access points and surveillance cameras. Data produced from such devices (*e.g.*, events generated at the access points when mobile devices carried by individuals connect, images captured by cameras when people enter/exit spaces) can be used to localize individuals within building at different levels of granularity. Recent work [60] has shown that while coarse-level region localization based on Wi-Fi connectivity can be achieved relatively efficiently, *e.g.*, ≈10ms/event,[1] ML techniques applied to the same data can help localize individuals at room-level granularity, at the expense of ≈200ms/event on same server. Likewise, video data with face detection & recognition (≈1s/image) could result in even higher accuracy. Now, consider analytical applications that perform tasks such as computing average occupancy level of a given location (meeting room), average time spent by people in the cafeteria during lunch, and number of people a person came in contact with during a time interval. Supporting such applications requires localizing people to appropriate rooms/regions based on WiFi and/or camera data. At rates of ≈1K WiFi events/sec and ≈100 images/sec, analyzing six months of data requires enriching 15 billion WiFi events and 1.5 billion images. Complete enrichment of data would require several years on a suitably large server, which is not feasible. Further, while coarse WiFi-based localization could be performed at data ingestion, fine-grained localization (using either WiFi or camera data) cannot be fully performed in its entirety (it would take ≈5 minutes of computation for one second of data generated requiring a large server farm). Such computations would be wasted if the analyst is satisfied by coarse localization for most parts of the building and decides to restrict attention to only a small subset of data (*e.g.*, data from atrium) for finer analysis. EnrichDB is designed to support real-time analysis in such scenarios when it is expensive/infeasible to enrich data in its entirety. ■

Similar scenarios as above exist in other domains too, *e.g.*, analysis of social media or web-traffic data using enrichment functions of topic detection, sentiment analysis, and network surveillance.

EnrichDB. EnrichDB, here after referred to as EDB, is a novel database technology that seamlessly integrates data enrichment inside data management. It extends the standard relational data model to empower users to declaratively associate enrichment functions with specific attributes during schema specification. Given an attribute, multiple enrichment functions could be associated either at the time of specifying a relation schema and/or later using schema modification operations. The execution of the enrichment functions is orchestrated by EDB. More specifically, based on enrichment policies, EDB may execute such functions during ingestion, intermittently on occurrence of certain events, or during query evaluation (based on the query). The declarative specification significantly simplifies development of analytical applications. Using EDB, an analyst can largely focus on developing/learning effective enrichment functions (that provide the requisite level of quality) instead of when and how such functions should be invoked. Furthermore, when an enrichment is performed during query execution, EDB exploits query semantics to reduce/eliminate redundant enrichment of objects that do not affect query results. For instance, in the motivating example above, if the analyst's interest is in occupancy of certain parts of the building (*e.g.*, meeting rooms) or

[1] On a server with 64 core Intel Xeon CPU E5-4640 with 2.40GHz clock speed and 128GB memory.

during certain hours (*e.g.*, daytime), EDB would restrict usage of expensive enrichment functions to only events/data that would influence the answer quality. While EDB restricts redundant enrichment during query processing, the execution (of non-redundant functions) can still result in long latency. In order to support (near) real-time analytics, EDB further supports an iterative approach, wherein query execution time is divided into multiple smaller time intervals, referred to as *epochs*. In each epoch, objects are selectively enriched, and a query answer is produced incrementally based on the enrichment performed till that epoch. Such a progressive approach provides several benefits: (*i*) it allows EDB to dynamically determine which objects to avoid enriching that could not have been avoided based only on static assessment of the query prior to execution, (*ii*) it enables EDB to rank objects based on assessing the impact of enriching them to query results, and (*iii*) it empowers users to stop execution at any time they wish as soon as a satisfactory level of quality is achieved.

Related Systems. While the focus of EDB on seamless integration of enrichment into data management is unique, it is based on several concepts that have been explored in the context of analyzing big data in the past. In particular, progressive query answering has also been explored in *online approximate query processing* (AQP) systems [29, 40, 90, 94] that improve aggregate query results over time. While AQP systems support progressive answers to deal with large data, EDB does so to hide latency due to expensive enrichment functions during query processing.

Another related technology to EDB is *Incremental-Materialized-Views* (IMV) as used in systems of DBToaster [56], LINVIEW [67], and PostgreSQL IMV [4], as well as works on *Intermittent Query Processing* (IQP) systems [82–84]. IQP systems support efficient ways of maintaining/computing delta answers to queries, while minimizing resource consumption of CPU and memory. Efficient delta computation has also been studied in *data flow systems* [9, 64, 66] that provide efficient state management techniques to speed up delta processing. While EDB exploits delta computation of queries (it is built using IMV and could benefit from work such as [82–84]), its focus is on incremental enrichment, which is complementary to the above methods.

Paper Outline. §2 describes the data and query model supported by EDB. §3 describes EDB architecture, implementations, and §4 describes the details of query processing. §5 discusses extensions to EDB to better support uncertainty resulting from enrichment function outputs. §6 provides a comprehensive evaluation of various design choices and techniques supported in EDB. EDB codebase is available at [1].

2 EDB DATA AND QUERY MODEL

Data Model. EDB follows the relational data model with the difference that some attributes can be specified as being ***derived***, and all other attributes are ***fixed*** attributes. A sample `CREATE TABLE` command below illustrates that the attributes `topic` and `sentiment` of `TweetData` table are derived attributes, while others are fixed.

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

The keyword `StateCutoff` used in the above statement will be explained in §3.2. Intuitively, it stores the attribute state in a compressed way for large domain attributes. The `feature` attribute contains term frequency-inverse document frequency (tf-idf) vector [58] pre-extracted from tweets. The value of a derived attribute is determined using one or more ***enrichment functions*** associated with it. We refer to the value of derived attributes as ***tags***.

Enrichment functions. An enrichment function takes a set of attribute values of a tuple as input and outputs either a single value (*i.e.*, binary classifier [81] that outputs *yes/no* answer for predicting a label), or multiple values (*i.e.*, multi-label classifier that outputs top-k predicted labels [18]), or a probability distribution (*i.e.*, probabilistic classifier that outputs a probability distribution over predicted labels [28]). Each enrichment function has an associated: (*i*) *cost*: that is the average execution time of the enrichment function on a single tuple, and (*ii*) *quality*: that is the metric of the goodness of the classifier (*i.e.*, accuracy) of the enrichment function in determining the correct value of the derived attribute. In general, the classifier quality depends upon the type of classifier.

The enrichment function could be either selected from a set of functions included in EDB (*e.g.*, multi-layer perceptron (MLP) and decision tree) or added as a UDF through a `CREATE FUNCTION` command. Enrichment functions are often classifiers (*e.g.*, sentiment predictor and object detector) that need a trained model. For such functions, EDB supports stored procedures (called `training procedures`) to train the model on a training dataset. This procedure takes as input a table containing the training dataset, set of features in the data on which the model needs to be trained, the target attribute, and the other parameters needed for training. The syntax of defining a `training procedure` is similar to Apache MADlib [41]. The output of the `training procedure` is a table containing the learned model.

Below, we show an example command to learn a model named `sentiment_mlp_model` for MLP-based sentiment detector using a training procedure named `mlp_train`. The model `sentiment_mlp_model` is trained based on a training data table called `tweets_train` that has `feature` column (fixed) and `sentiment` column (derived). The remaining arguments (*i.e.*, learning rate (`lr`), `tolerance`, `number of iterations`, activation function (`tanh`), and number of folds for cross-validation (`fold_num`)) are parameters for the `training procedure`. Likewise, there could be other functions based on random forest, decision tree, or naïve Bayes classification algorithm to compute `sentiment` values. Similarly, we can have functions to compute the `topic` (not shown in example). EDB supports a variety of function implementations apart from the one provided in MADlib.

```
CALL enrichdb.mlp_train('tweets_train','
    sentiment_mlp_model','feature','sentiment','
    layers=5,lr=0.003,iterations=500,tolerance=0,
    activation_function=tanh, fold_num = 3');
```

The *cost* and *quality* of enrichment functions can either be specified by user or can be determined automatically, (by using k-fold cross-validation during the training phase, as done in MADlib; based on the value of k, EDB splits the training data table into a training and validation dataset). EDB can also use a sample from the same training dataset used by `training procedure` provided

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`. `topic` & `sentiment` values are 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.

by the user as validation dataset. The *quality* of the function is computed by executing it on the validation dataset and measuring the *area under the curve* (AUC score) [21] (using the model learned on the training data). The *cost* of the function is the average execution time of the function on the validation dataset. The set of enrichment functions for a derived attribute $\mathcal{A}_i$ are called ***function-family*** of $\mathcal{A}_i$. The function-family of $\mathcal{A}_i$ is created using an `assign` function. We use ***calligraphic font*** for ***derived attributes***.

```
CALL enrichdb.assign('t_sentiment_family',
'TweetData','sentiment',[
['mlp_classifier','sentiment_mlp_model',0.1,0.86],
['dt_classifier','sentiment_dt_model',0.06,0.7]]);
```

Above, `assign` function creates a function-family named `t_sentiment_family` for `sentiment` attribute and contains functions *mlp_classifier*() and *dt_classifier*(). Function *mlp_classifier*() uses `sentiment_mlp_model` for sentiment prediction with *cost* = 0.1 second/tuple and *quality* = 0.86 (measured using AUC).

The outputs of enrichment functions in a function-family are combined using a ***combiner function***. EDB currently supports `weighted average` (default), `majority voting`, and `majority average` based combiner functions. Future version of EDB will support other complex combiner functions, *e.g.*, stacking [89] and boosting-based [38] combiners.

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 contains two components: ***state-bitmap*** that stores a list of enrichment functions that are already executed on $t_k.\mathcal{A}_i$; and ***state-output*** that stores the output of the executed enrichment functions on $t_k.\mathcal{A}_i$. For instance, 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 tags: d_1, d_2, and d_3. Thus, the state-bitmap for $t_k.\mathcal{A}_i$ contains $\langle 1010 \rangle$, 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 enrichment functions. Note that the first and third arrays contain values since f_1 and f_3 have been executed on $t_k.\mathcal{A}_i$. (Mechanisms to compress the state representation will be explained in §3.2.)

State of Tuples and Relations. The notion of the state of derived attributes is generalized to the state of tuples and relations in a straightforward manner. 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 $state(t_k) = \langle state(t_k.\mathcal{A}_p) || state(t_k.\mathcal{A}_q) || state(t_k.\mathcal{A}_r) \rangle$. EDB stores the tuple states in a different table (instead of inlining with t_k). The rationale behind this design choice is explained later in §3.2. The state of a EDB relation ($state(R)$) contains the states of all tuples of the relation (*i.e.*, $state(R) = \langle state(t_1) || state(t_2) || state(t_3) || state(t_4) \rangle$, where the relation R contains four tuples t_1, t_2, t_3, and t_4).

Query Model. EDB follows *iterative query execution model*, where query execution time is divided into smaller equi-sized time intervals, called *epochs* (e_i denotes the i^{th} epoch). The duration/length of an epoch is specified by users with queries. Epoch-based iterative implementation enables *progressive* return of query answers at the end of each epoch from partially enriched data and enables users to stop query execution at any time, while receiving high quality results. Epoch-based execution empowers EDB to adapt execution at epoch boundaries by selecting enrichment functions to execute.

To support the above execution strategy, EDB determines the value of the derived attribute based on its current state, and such attribute values could change across epochs as more enrichment functions execute, resulting in progressively improved answers. Below, we discuss: (*i*) the way to determine the attribute value given their states, (*ii*) semantics of query results over partially enriched data, and (*iii*) the strategy of incrementally computing query results across epochs.

Determinization Function. In EDB, the value of an attribute $t_k.\mathcal{A}_i$ is determined using a *determinization function* ($DET(.)$) based on its state $state(t_k.\mathcal{A}_i)$). $DET(state(t_k.\mathcal{A}_i))$ returns a single value for $\mathcal{A}_i$[2] or a NULL value, representing the situation when state of the attribute does not provide evidence to assign any value for $\mathcal{A}_i$.

Determinization concept naturally extends to a tuple and relation. The determinized representation of a relation R is denoted by: $DET(R) = DET(state(t_i.\mathcal{A}_j)) \mid \forall t_i \in R, \forall \mathcal{A}_j$ (derived attributes) of R.

Example 2.1. To illustrate the determinization process, consider a relation `TweetData` with two derived attributes `topic` and `sentiment` (see Table 1) and assume state-output of the two derived attributes, shown in Table 2. The `feature` attribute of Table 1 contains term frequency-inverse document frequency (tf-idf) vector [58] pre-extracted from tweets. We will explain in §3.2 how the state table (Table 2) is created. Based on Table 2 and a determinization strategy, we show the *tags* for `topic` and `sentiment` columns in Table 1. Note that Table 2 contains the possible values, whereas Table 1 contains the corresponding tags. ■

Query Semantics. Based on the definition of determinization function and the predicate evaluation logic of standard SQL, the query semantic for EDB is defined as follows:

$$q(R_1, R_2, \ldots, R_n) = q'(DET(R_1), DET(R_2), \ldots, DET(R_n))$$

where $q(R_1, R_2, \ldots R_n)$ is a query on relations $R_1, \ldots, R_n$, $DET(R_i)$ is the determinized representations of the i^{th} relation. Query q is rewritten as q' to be executed on the determinized representation of the relations using standard SQL logic.

Example 2.2. Consider a query (Query 1) on `TweetData` (Table 1) that retrieves tweets with `topic=social media`, `sentiment=positive`, and posted between 4 p.m. and 6 p.m. The query is executed on determinized representation of tuples as described in Example 2.1 returning the tuple t_1 of Table 1.■

[2] Generalized determinization functions that return multiple possible values for a derived attribute to improve answer quality [59, 68, 91] are also supported by EDB as discussed in §5.

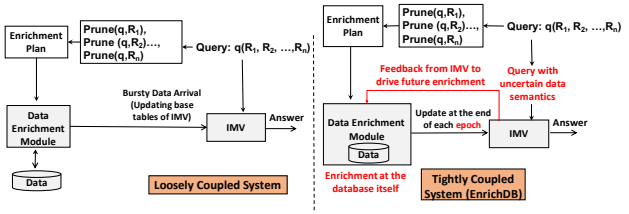


Figure 1: Comparison of EDB with loosely coupled system.

```
QUERY 1: SELECT Tweet,location,TweetTime, topic,
sentiment FROM TweetData WHERE sentiment='pos'
AND topic='soc. media' AND TweetTime
BETWEEN('16:00','18:00');
```

Progressive Query Execution. For producing query answers in epochs, EDB supports a mechanism to refine previously produced query answers, by retracting or adding tuples for set-based results or by improving the confidence intervals [40] 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 in EDB, users may stop query evaluation at any instance of time, performing enrichments that impact the answer quality as early as possible is desirable. EDB's effectiveness is measured using the following progressive score (similar to other progressive approaches [10, 11, 70]):

$$\mathcal{PS}(Ans(q, E)) = \sum_{i=1}^{|E|} W(e_i) \cdot [\mathcal{Q}(Ans(q, e_i)) - \mathcal{Q}(Ans(q, e_{i-1}))] \quad (2)$$

where $E = \{e_1, e_2, \ldots, e_k\}$ is a set of epochs, $W(e_i) \in [0, 1]$ 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 epoch e_i. In Equation 2, the quality $\mathcal{Q}$ of a set-based query answer can be measured using set-based quality metrics: precision, recall, F_α-measure [76], or Jaccard similarity coefficient [49]. The quality $\mathcal{Q}$ of an aggregation query can be measured using root-mean-square error [46] or mean-absolute-error [88]. Given a query, the maximum number of epochs, a quality metric for answers, and a set of weights assigned to epochs, the goal of EDB is to achieve the maximum progressive score for the query.

3 EDB ARCHITECTURE

EDB could be implemented using one of the following two approaches as presented in Figure 1: (*i*) a *loosely coupled* (LC) approach wherein an enrichment module is implemented separately from the DBMS, or (*ii*) a *tightly coupled* (TC) approach wherein enrichment

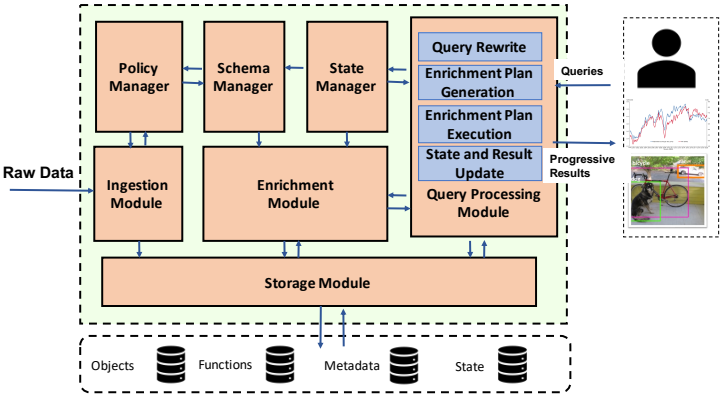


Figure 2: EDB Architecture.

is integrated more tightly with query processing inside the DBMS. EDB follows the TC approach, since it offers several additional opportunities for optimizations.

In either approach, on receipt of a query q, for each relation $R_i \in q$, EDB would analyze q to determine the subset, say $\mathbb{S}_i$, of R_i denoted by $Prune(q, R_i)$ that represents all tuples of R_i that could contribute to answers to q (details of $Prune(q, R_i)$ will be presented in §4.2). Such a set $\mathbb{S}_i$ of tuples in $\forall R_i \in q$ taken together represent the candidate set of tuples that may need enrichment for q.

In an LC approach, the set $\mathbb{S}_i$ is statically determined at the beginning of q, objects are incrementally chosen and enriched from this set, and the database is appropriately modified with the result of enrichment. The underlying DBMS can then use an incremental computation to determine the query results. In contrast, in the TC approach, the candidate set of objects (*i.e.*, $\mathbb{S}_i$) for enrichment is continuously modified with additional feedback from the incremental answers generated to further prune and/or prioritize objects that may need to be enriched to answer q.

For instance, consider a query that retrieves tweets in Table 1 with `sentiment = 'negative' AND topic = 'social media'`. Say values of both t.`topic` and t.`sentiment` were NULL for a tuple t. In the LC approach, all the functions associated with `sentiment` and `topic` attributes would be in the candidate set of enrichment to be performed on t. While the LC approach will result in a possible execution of all such enrichment, a feedback from the query engine during execution that t does not meet the condition of `sentiment = 'negative'` may save on execution of topic-based enrichment functions. Such a pruning strategy can be very effective, when queries are more complex and are selective, as illustrated in §6.2. TC approach offers other additional advantages in terms of executing enrichment functions closer to data (in the database engine), and, as discussed in §5, allows us to incorporate additional semantics to improve query result.

Figure 2 presents the architecture of EDB that is implemented as a layer on top of a relational database (*i.e.*, PostgreSQL).[3] In EDB, layering consists of appropriate translation of data types to PostgreSQL tables and association of enrichment functions with user-defined functions (UDFs). EDB stores enrichment policies, enrichment state of tuples, and the metadata in the form of PostgreSQL tables that are used during ingestion and query processing.

[3]Layering on top of SparkSQL is an ongoing work. and we present a related experiment in §6.2.

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 3: `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 4: `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 5: `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 6: `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 7: `TweetDataState` table (created for tuples in `TweetData` table).

To avoid confusion, we will use the term **EDB table (or relation)** and **EDB query** to distinguish them from PostgreSQL tables and PostgreSQL queries to which they are mapped.

We focus primarily how EDB performs data enrichment during query execution. The key challenge is to design and implement a strategy that progressively enriches data using diverse functions while executing queries. This resulted in a set of design decisions taken by us as described in this section. This section describes schema manager §3.1, state manager §3.2 in detail, and briefly discusses the functionalities of the remaining modules §3.3.

3.1 Schema Manager

EDB schema manager maintains metadata about EDB 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). EDB catalog consists of the following components:

Metadata of EDB relations. `RelationMetadata` stores names of each EDB 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 3.

Metadata of enrichment functions. It is stored in `FunctionMetadata` table. EDB supports three types of functions: (*i*) functions implemented in Python, using standard ML libraries, *e.g.*, scikit-learn [74] and TensorFlow [6], (*ii*) analytics library of Apache MADlib [41], 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 EDB 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 5). Some of these functions require a training model. If a model is trained using EDB `training procedure`, the model is stored in a `model_table`. For such functions, `FunctionMetadata` table stores `model_table` name. EDB's `model_table` follows a similar format as in MADlib (`model_table` is discussed in Appendix 9.1).

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 4.

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 EDB, 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 `DecisionTables` that differ based on the mapping functions used to guide the enrichment process in EDB, by default, EDB uses a mapping function that — given a state of the attribute — determines the reduction in entropy [43] 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 $D_1, \ldots, D_n$ with probabilities $p_1, \ldots, p_n$, entropy is measured as: $\sum_{i=1}^{n} -p_i \log p_i$.

Table 6 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 7) 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 6, since entropy of t_1 is in the range (0.25-0.5), EDB determines the next best function to execute on t_1 is f_3 and its benefit as 0.2.

Learning of mapping function. EDB 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, EDB 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 6). 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 EDB supports learning of the above map based on entropy, users can provide their own custom maps to guide the selection of functions that EDB uses for further enrichment. Alternatively, EDB 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 7) that is created at the time of creation of EDB 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.

EDB stores the state of tuples as a separate table instead of inlining with EDB tables, since the size of the State table is often much smaller compared to EDB 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 7) 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, EDB uses a compressed representation to store StateOutput of tuples for attributes with large domain sizes. For such attributes, EDB 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 [55] 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.

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 8: Enrichment policies.

3.3 Other Modules

Policy manager: maintains a set of *enrichment policies*. A policy consists of a predicate against which the tuples are matched, a corresponding action (*i.e.*, a list of enrichment functions) that should be executed, the priority of the policy, and when the policy is applicable (ingestion time, in the background after ingestion or query time, which is the default). If multiple policies apply to a tuple, actions specified in the policy with higher priority are executed. If priorities are equal, actions in the same function-family are executed based on expected quality. Policy manager provides APIs to insert/delete/update policies. Table 8 shows example policies on TweetData relation, where, the first policy states that all tuples of TweetData with TweetText containing the keyword 'covid' need to be enriched using *mlp_classifier*() function at ingestion time. In the current implementation, policies are specified by the user/domain expert. Future versions will explore a self-driving approach to learn policies based on query workload.

In the current implementation, policies are specified by the domain expert. Future versions will explore a self-driving approach to learn policies based on query workload.

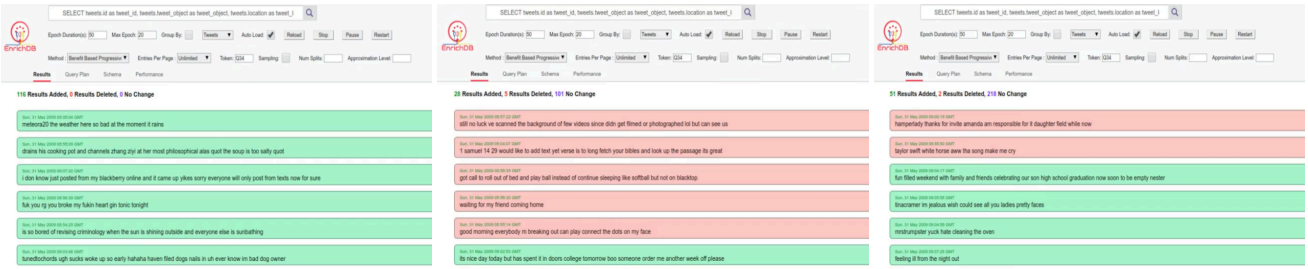


Figure 3: EDB web interface. The images show the query results at the end of three epochs during query execution. Green box shows newly added tuples and red box shows deleted tuples from previous epoch.

Data enrichment module: supports mechanisms to create, delete, store, and execute enrichment functions. For functions that require learning a model, it provides APIs to add new `training procedures`, as described in §2.

Data ingestion module: provides APIs to ingest data through streams or data stored as CSV files. Data from various data sources reach the data ingestion module. Given the type of incoming tuples (*e.g.*, tweets, data from WiFi access points, or images from cameras), data ingestion module interacts with schema manager and policy manager to check the enrichment policies applicable to them. If an enrichment policy applies to a tuple, the data enrichment module executes enrichment functions on the tuple depending on the policy. Finally, the tuples and the enrichment function outputs are sent to the storage manager and are stored in PostgreSQL tables.

Storage manager: stores EDB tables, associated metadata, and the states of the tables in the underlying DBMS. Storage manager provides APIs to enable other modules to store, update, and fetch data into/from the underlying database system without requiring any knowledge of the specific database system used.

3.4 User Interfaces

A user interacts with EDB 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 EDB 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.

EDB web application. EDB 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, EDB 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 3. 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 [40]. The only difference is that a `GROUP BY` key can be added or removed in a later epoch.

4 PROGRESSIVE QUERY PROCESSING

Query processing in EDB consists of four main steps (Figure 4): 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 EDB sets up tables used during the enrichment plan generation phase (will be clear in §4.2). An EDB query is wrapped in a stored procedure, called *EDB 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, EDB includes the following UDF to support determinization of tuples (as described in §2).

Determinization UDF (*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 5(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

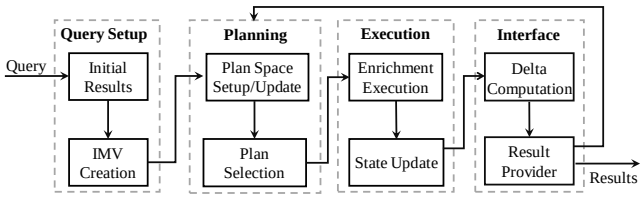


Figure 4: Query processing stages in *executor* procedure.

query is shown in Figure 5. The query rewriting process contains the following stages:

(1) In order to find relations that require determinization, EDB lists all relations that have query predicates/projection on derived attributes (*e.g.*, R and S in our example Figure 5).
(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, the 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 5), and do not need determinization.

Example 4.1. We show the query rewrite strategy of EDB for the query shown in Query 1 on `TweetData` table. The following SQL query is the rewritten query of EDB *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,
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 EDB. Thus, below, we describe several optimizations performed by EDB during the query rewrite step. EDB 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.** EDB pushes down the selection condition on fixed attributes in the query tree. Consider a query shown in Figure 5(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 EDB is shown in Figure 5(b). For more complex selection conditions, EDB 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 EDB 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

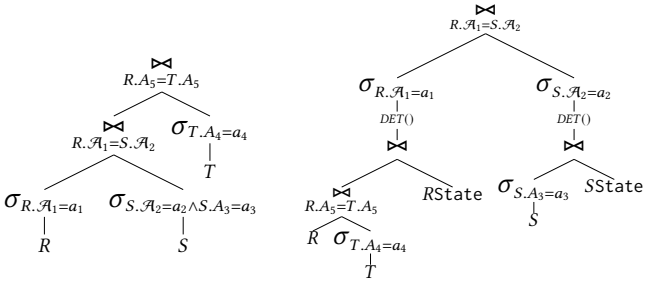


Figure 5: (a) Original Query (lhs) (b) Rewritten Query (rhs).

of Figure 5(a), where join condition (*i.e.*, $R.A_5 = T.A_5$) is on a fixed attribute. In the rewritten query (see Figure 5(b)), EDB 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 EDB 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, EDB creates an incremental materialized view (IMV) [5, 19, 56, 67] 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_K) consists of a set of ⟨tuple, enrichment function⟩ pairs, where the enrichment function will be executed on the tuple in e_K. These plans are selected at the beginning of each epoch using either benefit- or sampling-based plan generation method. The ***benefit-based*** (**BB**) method selects pairs of tuples and enrichment functions based on a specific `DecisionTable` (learnt by EDB or provided by user as discussed in §3.1), while the ***sampling-based*** (**SB**) method *randomly* selects ⟨ tuple, enrichment function ⟩ pairs. The effectiveness of a plan generation strategy is measured using the progressive score defined in Equation §2 (§6 experimentally compare these plan generation methods). Importantly, in EDB, users can incorporate their own plan generation methods by implementing functions *GetNextFunction*() and *SelectBestFeasiblePlan*(), shown in Algorithm 1.

EDB 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

Rel.	TID	Attr.	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.

Rel.	TID	Attr.	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.

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 6-15): This step is executed *only in the first epoch* to create and initialize PlanSpaceTable. Given q, an entry (*i.e.*, ⟨tuple ID, next best function to be executed⟩) 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 11). In SB method, *GetNextFunction*() randomly selects a function that is not yet executed on the tuple (Line 13).

Plan Selection (Lines 16-20). 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 [22], which is NP-hard. Thus, we use Weighted Shortest Processing Time first (WSPT) heuristic-based approach [22] to select a set of tuples from PlanSpaceTable. Using WSPT, EDB, 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 sample of tuples from PlanSpaceTable. While any sampling can be used (*e.g.*, uniform random, stratified random, or variational sub-sampling as used in [7, 73]), EDB uses uniform random sampling.

Plan Space Update (Lines 21-27). Since the execution of a plan in an epoch alters the state of the data, EDB 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, EDB updates only those tuples of

Algorithm 1: EDB plan generation.

Inputs: $\langle q, \delta, \texttt{state}(R_i), type, \text{DT}, \kappa \rangle$. // q: a query on relations $R_1, \ldots, R_n$, δ: epoch duration, *type*: type of plan generation - sampling or benefit, DT: DecisionTable, and κ: epoch id.
Outputs: Enrichment plan, *i.e.*, PlanTable, for q for epoch κ.
Variables: f_m: enrichment function. ***benefit***: benefit of enrichment.

1 **Function** ***GeneratePlan*(κ) begin**
2 **if** $\kappa = 1$ **then** PlanSpaceTable ← *PlanSpaceSetup*(*type*)
3 **else** PlanSpaceTable ← *PlanSpaceUpdate*(*type*)
4 PlanTable ← *SelectBestFeasiblePlan*(*type*,PlanSpaceTable,δ)
5 Return PlanTable
6 **Function** ***PlanSpaceSetup*(*type*) begin**
7 **for** $\forall R_i \in q$ **do**
8 *attribute_list*[] ← *GetDerivedAttributeQuery*($q(R_i)$)
9 **for** $\forall (j, \ell) \in [\mathcal{A}_j \in attribute\_list[]] \times [t_\ell \in R_i]$ **do**
10 **if** *type* = benefit **then**
11 f_m, *benefit* ← *GetNextFunction*(state($t_\ell.\mathcal{A}_j$), DT)
12 Append $\langle Name(R_i), ID(t_\ell), Name(\mathcal{A}_j), f_m, benefit \rangle$ to PlanSpaceTable
  else
13 f_m ← *GetNextFunction*(state($t_\ell.\mathcal{A}_j$))
14 Append $\langle Name(R_i), ID(t_\ell), Name(\mathcal{A}_j), f_m \rangle$ to PlanSpaceTable
15 Return PlanSpaceTable
16 **Function** ***SelectBestFeasiblePlan*(*type*, PlanSpaceTable, δ) begin**
17 **if** *type* = benefit **then**
18 PlanTable← *GetTopTuples*(*Sort*(PlanSpaceTable, *benefit*), δ)
19 **else** PlanTable← *GetRANDSample*(PlanSpaceTable, δ)
20 Return PlanTable
21 **Function** ***PlanSpaceUpdate*(*type*) begin**
22 **for** $\forall \langle Name(R_i), ID(t_\ell), Name(\mathcal{A}_j), f_m, benefit \rangle \in$ PlanTable **do**
23 **if** *type* = benefit **then** f_p, *benefit* ← *GetNextFunction*(state($t_\ell.\mathcal{A}_j$), DT)
24 **else** f_p ← *GetNextFunction*(state($t_\ell.\mathcal{A}_j$))
25 **if** $f_p = \emptyset$ **then** Delete $\langle Name(R_i), ID(t_\ell), Name(\mathcal{A}_j), f_m, benefit \rangle$ from PlanSpaceTable
26 **else** Update the function for $Name(R_i), ID(t_\ell), Name(\mathcal{A}_j)$ to f_p in PlanSpaceTable.
27 Return PlanSpaceTable

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 23) 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 25).

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. EDB 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, EDB attempts to find the smallest such subset. To compute $Prune(q, R_i)$, EDB 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

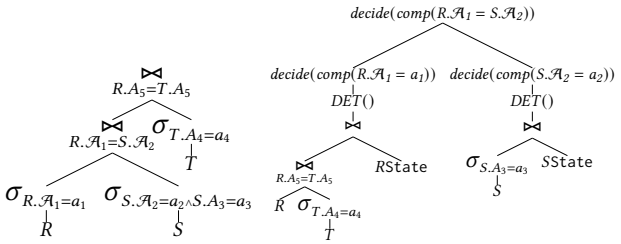


Figure 6: (a) Original Query(lhs) (b) Rewritten Query(rhs).

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 5, 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 & State Update

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 EDB. 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, EDB 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.

Relation	#tuples	Size(GB)	Derived attrs.	Functions used
TweetData	11M	10.5	sentiment(3)	GNB,KNN,SVM,MLP
			topic(40)	GNB,KNN,LDA,LR
ImageNet[34]	100K	19	ObjectClass(100)	DT,GNB,RF,MLP
MultiPie[80]	100K	16.9	gender(2), expression(5)	DT,GNB,KNN,MLP
SyntheticData	100M	35	$\mathcal{A}_1$(5)	$f_1, f_2, \ldots, f_{10}$.

Table 11: Datasets used in experiments.

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, EDB allows users to retrieve delta changes of the answers, *i.e.*, inserted/deleted/updated tuples from the previous epoch. EDB provides the delta answers to users through the delta tables that are maintained by IMV. Delta tables store the delta results that are computed from a delta relational algebra expression [4, 67] 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 EDB is not demand-driven, as in SQL databases.

5 HANDLING UNCERTAINTY

Enrichment functions such as ML classifiers can output a probability distribution over a set of possible values. EDB, as discussed so far, converts such distributions into a single value using a determinization function and executes queries on resulting relations. As shown in prior research [59, 68, 91], such a determinization strategy may result in suboptimal answer quality, *e.g.*, it forces a hard decision between two values even with minute differences in their score from a classifier. The alternate is to allow the underlying data to represent/maintain uncertainty (*e.g.*, probability distribution over possible values) and thus return objects that may not be part of the answer in the top-1 determinized representation (*e.g.*, most probable database instance).

Prior work has considered several models for handling uncertainty. A commonly used approach is possible world semantics over probabilistic databases [8, 32, 45, 51, 78] where the query is (conceptually) executed over all possible worlds and the answer consists of a list of ⟨tuple, confidence⟩ pairs, where confidence is the sum of probabilities of all worlds in which the tuple was present in the answer. Alternate approaches include consensus world semantics [59], generalized determinization to optimize quality metrics [59, 68], and uncertainty annotated databases [36], where attribute values are represented as a set, any value of which could be the attribute value, following the *OR*-semantics [47, 48].

EDB could support possible world semantics, by layering it on top of probabilistic databases, such as MayBMS [45], Trio [8], or MCDB [15, 50]. However, efficient implementation of such an approach will require computing incremental materialized views over probabilistic databases. To the best of our knowledge, such techniques have never been explored in the past in the literature. In contrast, model such as *OR* query semantics where a determinization function returns a set of one or more values, which the attribute can take, is simpler to implement in EDB.

In order to support *OR* query semantics with determinization functions returning a set of values, EDB extends the three-valued

condition evaluation logic of SQL to four-valued logic. The truth values in four-valued logic are: **true (*T*), false (*F*), possible (*P*), and unknown (*U*)**. Here, *P* represents that the condition is **possibly true**, whereas *U* represents that the truth value is **unknown** based on current state of enrichment. Like SQL, EDB does not return tuples that evaluate to *U* in the answer, while tuples evaluating to *P* may or may not be returned, depending on maximization of progressive score of the query, as defined in Equation 2 in §2. Below, we discuss how EDB assigns truth values to predicates/expressions.

• ***Simple predicates*** are formed using expression of the form $\mathcal{A}_i$ op a_m, where op is one of the following comparison operators: $\langle =, \neq, >, \geq, <, \leq \rangle$, and $\mathcal{A}_i$ is a derived attribute. If $DET(state(t_k.\mathcal{A}_i))$ is NULL, then, as in SQL, expression evaluates to U. If $DET(state(t_k.\mathcal{A}_i))$ is a singleton set (say S) and $\exists x \in S$ such that x op a_m holds, then the expression evaluates to T; otherwise, evaluates to 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 $\nexists x \in S$ such that x op a_m holds, then expression evaluates to F.

Consider an expression $\mathcal{A}_i$ **op** $\mathcal{A}_j$, where $\mathcal{A}_i$ and $\mathcal{A}_j$ are derived attributes 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 predicate evaluates to U. If the outputs of both $DET(state(t_k.\mathcal{A}_i))$ and $DET(state(t_l.\mathcal{A}_j))$ are singleton and the elements $x \in DET(state(t_k.\mathcal{A}_i))$ and $y \in DET(state(t_l.\mathcal{A}_j))$ are such that x op y holds, then the predicate evaluates to T; otherwise, F. In case one or both the outputs 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 predicate evaluates to P; otherwise, F.

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

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

• ***Aggregation.*** Aggregation conditions 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 the 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 a query that counts number of tweets with positive sentiment from `TweetData` of Table 1, and assume that Table 1 has 250 tuples of which 100 evaluated to T, while 20 of remaining 150 tuples evaluated to P. Here, EDB returns a range of $[100, 120]$. Likewise group-by aggregation results in one such range per group.

Implementation Details. To support above-mentioned four-valued logic, EDB implements the following two UDFs, in addition to the *DET* UDF, as explained in §4.

ID	Query
Q1	SELECT * from **ImageNet** where ObjectClass=2 where ImageID between (20000,30000)
Q2	SELECT * from **MultiPie** where gender=1 and CameraID <12
Q3	SELECT tid, UserID, Tweet, location, TweetTime from **TweetData** where sentiment = 1 and TweetTime between('16:00','18:00')
Q4	SELECT * from **SyntheticData** where $\mathcal{A}_1$=1 and $A_2 < 100000$
Q5	SELECT * from **MultiPie** where gender = 1 and expression =2 and CameraID <12
Q6	SELECT * from **TweetData** T1, **TweetData** T2 where T1.sentiment = T2.sentiment and T1.TweetTime between('16:00','18:00') and T2.TweetTime between ('16:00','18:00')
Q7	SELECT * from **TweetData** T1, State S where T1.location = S.city and S.state='California' and T1.sentiment = 1 and T1.TweetTime between('16:00','18:00')
Q8	SELECT gender, count(*) from **MultiPie** where CameraID <12 group by gender
Q9	SELECT ObjectClass, count(*) from **ImageNet** where ImageID between (20000,30000) group by ObjectClass

Table 12: Queries used.

Comparator UDF (*comp*()): takes a query predicate and determinized representation of the tuple as input, and outputs a truth value either T, F, P or U based on the evaluation logic of four-valued logic. *E.g.*, consider tuple t_1 of `TweetData` and query of Query 1. Suppose determinized representation of t_1 for `topic` is a set: {social media, entertainment, politics}, then *comp*(`topic` = social media) will output P as the truth value.

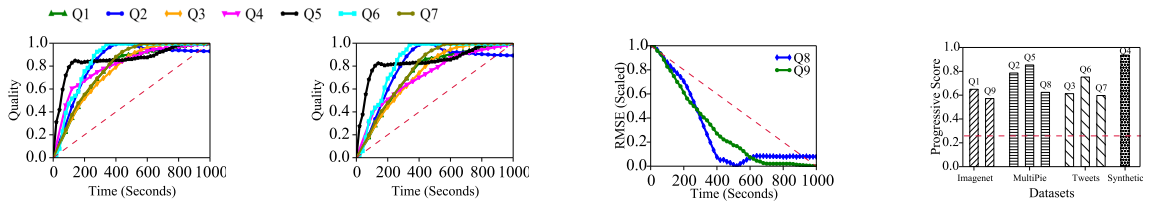
Decider UDF (*decide*()): takes a tuple and the output of *comp*() UDF and either passes the tuple to the next query operator or filters out the tuple. This decision is made only for the tuples that evaluate to P from *comp*() UDF. The other tuples that evaluate to T are passed to the next operator, while tuples with F or U are removed. Using the UDFs, the rewritten query is shown in Figure 6.

6 EXPERIMENTAL EVALUATION

We evaluate EDB from following perspectives: progressive quality improvement achieved during query execution and performance overhead of EDB due to iterative plan generation during epochs. For measuring progressiveness, we evaluate the impact of different plan generation strategies, the nature of enrichment functions, specifically the rate at which functions improve quality per unit cost. For performance, we study the impact of various optimizations.

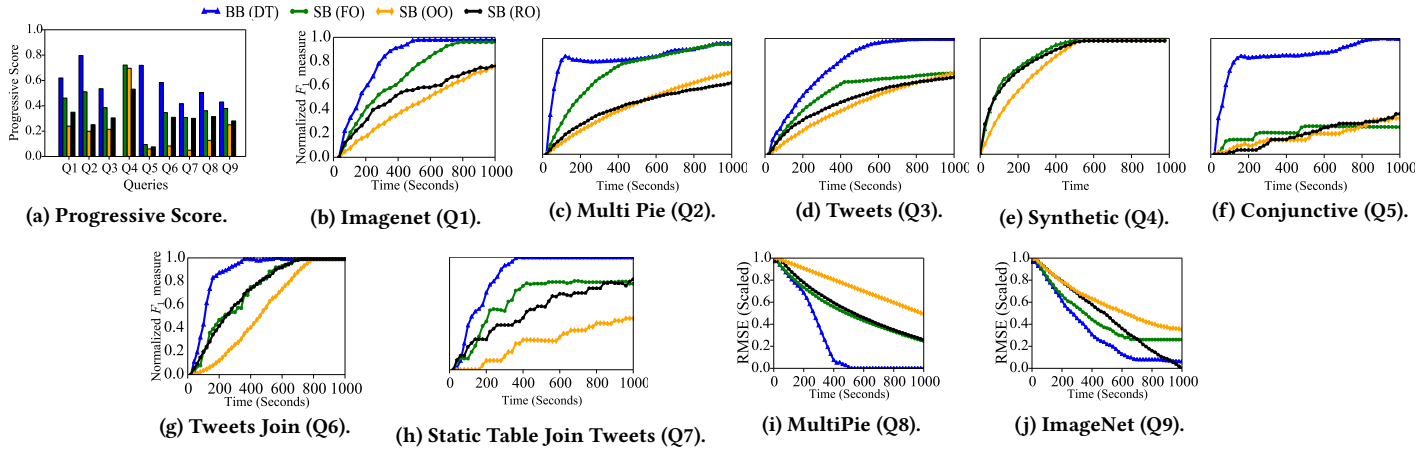
6.1 Experimental Setup

Datasets. We used four datasets to evaluate the performance of EDB. The datasets corresponded to:(*i*) TweetData collected using Twitter APIs containing 11 million rows (size = 10.5 GB) and derived attributes of `sentiment` (dom. size = 3) and `topic` (dom. size =40), (*ii*) ImageNet [34] dataset consisting of 100K images of objects (size = 19 GB) and derived attributes of `ObjectClass` (dom. size =100), (*iii*) MultiPie [80] dataset contains 100K facial images (size = 16.9 GB) with derived attributes `gender` (dom. size = 2) and `expression` (dom. size = 5), and (*iv*) a large synthetic dataset with 100M tuples (size = 35 GB) with a derived attribute $\mathcal{A}_1$ (dom. size = 5) for evaluating the scalability of EDB. Though EDB is deployed to build IoT applications in our campus, we restricted our experiments to publicly available datasets and functions. The campus dataset



(a) F_1 measure. (b) Jaccard's similarity. measure. (c) RMSE. (d) Progressive Score.

Figure 7: Progressiveness Achieved. The dotted line shows a possible incremental strategy producing query answers using server-side cursors.



(a) Progressive Score. (b) Imagenet (Q1). (c) Multi Pie (Q2). (d) Tweets (Q3). (e) Synthetic (Q4). (f) Conjunctive (Q5).

(g) Tweets Join (Q6). (h) Static Table Join Tweets (Q7). (i) MultiPie (Q8). (j) ImageNet (Q9).

Figure 8: Performance results of different plan generation strategies in EnrichDB.

contains private location information that is not available without institutional review board (IRB) approvals, and thus, cannot be shared with the larger community.

Enrichment Functions. In the real datasets, we used the following probabilistic classifiers: Gaussian Naïve Bayes (GNB), Decision Tree (DT), Support Vector Machine (SVM), k-Nearest Neighbor (KNN), Multi-Layered perceptron (MLP), Linear Discriminant Analysis (LDA), Logistic Regression (LR), and Random Forest (RF); as enrichment functions. The GNB classifier was calibrated using isotonic-regression model [92] and all the other classifiers were calibrated using Platt's sigmoid model [75] during cross-validation. After calibration, each classifier outputs a probability distribution [75]. For the synthetic dataset, we used synthetic functions each associated with varying levels of quality and cost. For synthetic functions, quality 0.8 means predicting correct values 80% of the time. We used `weighted average` as the combiner function, where weights are proportional to the quality of enrichment functions learned based on the mechanism of §2.

Queries. As shown in Table 12, we selected nine queries, where ***Q1-Q4*** are ***selection queries***, ***Q5*** is a ***conjunctive query***, ***Q6-Q7*** are ***join queries***, and ***Q8-Q9*** are ***aggregation queries*** with number of groups as 2 (low) and 100 (high) respectively. The ***epoch size*** for all queries in all experiments (except Exp 5) was ***20 seconds***.

Plan Generation Strategies. For experiments, four different plan generation strategies were used: (*i*) *Benefit-based approach using Decision Table* (**BB(DT)**): that selects a set of ⟨tuple, function⟩ pairs with the highest benefit value based on the decision table. (*ii*) *Sample-based strategy with Object Order* (**SB(OO)**): that randomly selects tuples from the set of tuples satisfying predicates on fixed attributes. Selected tuples are completely enriched by executing all enrichment functions available for derived attributes present in the query. (*iii*) *Sample-based strategy with Function Order* (**SB(FO)**): that selects enrichment functions based on the decreasing order of their $\frac{quality}{cost}$ values, as described in §2. The top-most function from the sorted enrichment functions is executed on all tuples (obtained after checking the predicates on fixed attributes), before executing the next function. (*iv*) *Sample-based Random Order* (**SB(RO)**): that selects both tuples and enrichment functions randomly from a set of ⟨tuple, function⟩ pairs after checking predicates on fixed attributes.

6.2 Experimental Results

The experiments were performed on an AWS server with 16 core 2.50GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD. **Exp 1-2** are progressiveness experiments, **Exp 3-4** are performance experiments and **Exp** 5 is an experiment for query parameter selection.

Exp 1: Progressiveness of different queries. Figure 7 evaluates EDB in terms of progressive quality improvement achieved for different types of queries (Q1-Q9). For real datasets, we used the BB(DT) method, whereas for synthetic dataset we used SB(FO) plan generation method. Figure 7(a) shows the results for set based queries Q1-Q7, where the quality of answers is measured using ***normalized F_1*** *i.e.*, F_1/F_1^{max}, where F_1^{max} is the maximum F_1 measure achieved during the query execution. The ***normalized Jaccard's similarity*** results are shown in Figure 7(b). For aggregation queries

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Time	31	44.5	22.1	14	67.1	39.2	22.8	45.1	31.8

Table 13: Time without prog. execution (in minutes).

Q8 and Q9, the quality is measured using ***normalized root mean square error*** (RMSE); Figure 7(c). We plot normalized measures as a function of time to emphasize the rate at which EDB improves the quality of query results across different queries and datasets instead of the actual F_1-measures. Actual F_1-measure varies across different queries (that belong to different datasets) based on the quality of classifiers chosen for enrichment (*e.g.*, the maximum F_1 measures for different queries were: for Q1 0.54, Q2 0.78, Q3 0.56, Q4 1, Q5 0.4, Q6 0.58, Q7 0.52). From Figures 7(a) and 7(b), we observe that EDB achieves a high quality improvement (95% of maximum F_1 measure and 95% reduction of RMSE) within the first few epochs of query execution. Also, EDB performs much better than a possible iterative strategy, where a database system chooses a tuple, completely enriches it, and if it satisfies the query predicate, returns it as an answer to the user (shown as the dotted line in Figure 7). Since for queries with blocking operators (joins and group-by aggregation such as queries Q6-Q9) all tuples need to be enriched completely before evaluating join or aggregation operators, such strategy cannot be devised.

Figure 7(c) shows the progressive score (as defined in Equation 2) achieved for each query. Observe that EDB achieves a high progressive score for all queries as compared to the possible iterative strategy (dotted line in Figure 7(c)). Table 13 shows the query execution time for Q1-Q9, when they are executed after complete enrichment of tuples. Comparing Table 13 with Figure 7, we observe that EDB returns high quality results within a few epochs without requiring users to wait for complete enrichment.

Exp 2: Effect of Different Plan Generation Strategies. Figure 8 studies different plan generation strategies and their impact on progressiveness. Figure 8(a) plots progressive score of all nine queries Q1-Q9. Figures 8a - 8j show BB(DT) performs better than sampling based approaches highlighting that the decision table learned by EDB using the validation dataset accurately represents the benefit of the chosen execution order of enrichment functions. Among sample-based strategies, SB(FO) performs the best and SB(OO) performs the worst since function order chooses functions with highest quality per unit cost before other functions. SB(RO) performs only marginally better than SB(OO). In the following experiments, we use BB(DT) as a plan generation strategy, except for synthetic dataset for which we do not store benefit values.

Exp 3: System Overhead. This experiment measures the following two overheads incurred by EDB:

(***i***) ***Time overhead***: measures the amount of time spent in *non-enrichment tasks* (*i.e.*, query setup, benefit calculation, plan selection, delta computation, and state update) to compare against the time involved in data enrichment. Figure 13 shows that the time overhead is significantly lower than the time spent in enrichment. Particularly, across all epochs, the total time in query setup, benefit calculation, plan selection, delta computation, and state update took at most 3s, 90s, 4s, 5s, 17s, respectively, while the total time spent across all epochs in enrichment was 1000s. Majority of the time for benefit calculation was spent in the first epoch due to the calculation of benefit of all tuples satisfying query predicates on fixed attributes.

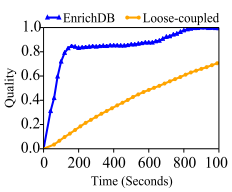


Figure 9: Comparison with LC (Q5).

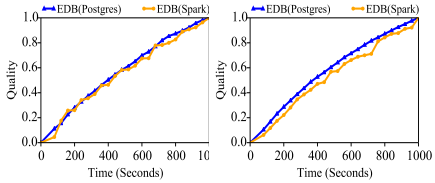


Figure 10: EDB (PostgreSQL) and EDB (Spark) implementation for (a)Q1,(b)Q2.

(***ii***) ***Storage overhead***: measures the size of all temporary tables and IMV, used during query processing to compare against the size of EDB tables. The maximum storage overheads of the benefit table, plan table, and IMV at any epoch for the queries of Q1-Q9 were 1.48 MB, 56 KB, and 1.2 MB respectively (see Table 14). Also, the combined size of all metadata tables, (*i.e.*, `RelationMetadata`, `FunctionMetadata`, `FunctionFamily`, and `DecisionTable`) is less than 10 MB. These overheads are significantly lower than the size of EDB tables. The state table sizes for TweetData, Imagenet, Multi-Pie and Synthetic datasets were 2.4GB, 6.8GB, 101MB and 246MB respectively (as shown in Table 15) which are much smaller than the corresponding EDB tables. Furthermore, using state cutoff (§3.2) strategy, state storage overhead reduced significantly. For ImageNet the state cutoff representation reduced the size of state table from 6.8 GB to 50 MB. For TweetData the state overhead reduced from 2.4 GB to 0.9 GB. The improvement was more in ImageNet because the `ObjectClass` had a larger domain of size 100 as compared to `topic` attribute with domain size of 40.

Query	PlanSpaceTable	PlanTable	IMV
Q1	536 kB	48 kB	168 kB
Q2	776 kB	56 kB	232 kB
Q3	816 kB	48 kB	1168 kB
Q4	840 kB	36 kB	216 kB
Q5	1488 kB	8.2 kB	48 kB
Q6	264 kB	56 kB	112 kB
Q7	824 kB	48 kB	126 kB
Q8	832 kB	48 kB	0.25 kB
Q9	528 kB	48 kB	12 kB

Table 14: Max. storage overhead.

Relation	State overhead	Overhead after state cutoff
TweetData	2.4 GB	936 MB
ImageNet	6.8 GB	50 MB
Multi-Pie	101 MB	-
Synthetic	246 MB	-

Table 15: Storage cost of state.

Query	#rows in `PlanSpaceTable`	#rows in `PlanSpaceTable` with optimization
Q1	100K	10,000
Q7	11M	50,000

Table 16: Impact of optimization.

Exp 4: Impact of Optimizations. §4.1 presents two re-rewrite optimizations to reduce the complexity of plan generation step and enrichment. We selected Q1 (filter on fixed attribute) and Q7 (join on fixed attribute) to evaluate these optimizations. The results are presented in Table 16, that shows that the tuples considered for enrichment were reduced significantly due to these optimizations. Furthermore, due to the reduction of benefit table size, the complexity of the enrichment plan generation phase reduced significantly.

Exp 5: Effect of Epoch Size. Figure 14(a) plots time to reach (TTR) 90% quality for Q2 with respect to epoch size. As epoch size reduces from 50 to 20, TTR reduces since smaller epoch size causes more frequent plan generation resulting in accelerated improvement of answer quality. Reducing epoch size further (*i.e.*, 5 or 10) results in increase in TTR since increased overhead of frequent plan generation begins to overshadow improvements in quality achieved (Figure 14(a)). Figure 14(b) shows the overhead of plan generation as a function of epoch size. The effect of epoch size to other queries (besides Q2) is similar.

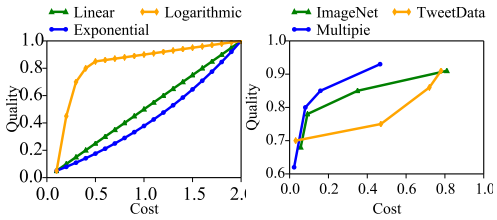


Figure 11: Cost vs. Quality (a) synthetic(lhs) and (b) real (rhs) functions.

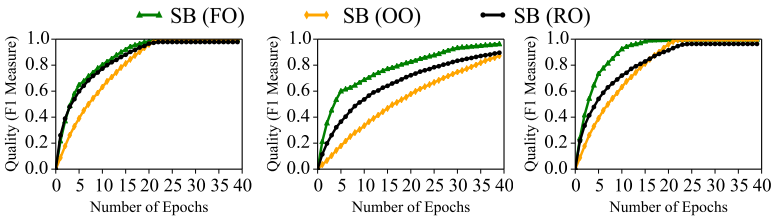


Figure 12: Comparing plan generation on synthetic data. (a) linear (b) logarithmic (c) exponential correlations.

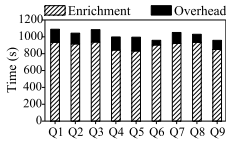


Figure 13: Time overhead.

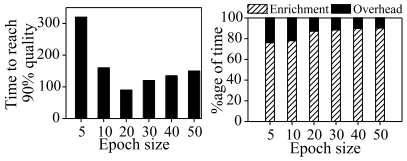


Figure 14: Effect of epoch sizes (a) TTR 90% (lhs) (b) overhead (rhs).

Exp 6: Comparison with Loosely Coupled Approach. This experiment compares EDB with the loosely coupled(LC) approach of enrichment. In LC approach, EDB performs enrichment plan generation only once. However, quality of answer is still measured at the end of each epoch. Figure 9 shows that the quality improvement of query results in LC approach is much lower than EnrichDB. The reason behind the performance gap is adaptive improvement of enrichment plan using feedback from the incremental answers generated by IMV as described in §3.

Exp 7: EDB implementation on Spark. This experiment shows the performance of EDB when it is implemented on top of Apache Spark. We implemented the logic of progressive query execution and epoch-based enrichment inside a Spark script [1]. For enrichment plan generation, we implemented the SB(OO) strategy of EDB on Spark due to its simplicity. The results shown in Figure 10 illustrate that it is possible to implement EDB on Spark.

Exp 8: Impact of enrichment function correlations. This experiment evaluates the impact of variations in cost and quality of different enrichment functions on the sample based plan generation strategies (*i.e.*, SB(OO), SB(FO), SB(RO)) of EnrichDB. Note that, we do not experiment with BB(DT) strategy, since creating a decision table for the synthetic functions would be artificial and hence would not provide much insight. For this experiment, we used 10 synthetic functions that have the following correlations in their cost and quality (shown in Figure 11): (*i*) *linear correlation* where the quality and cost of functions vary linearly, (*ii*) *exponential correlation* where the quality increases exponentially with respect to the cost (the enrichment functions used in `TweetData` follow such correlation), and (*iii*) *logarithmic correlation* where quality increases logarithmically with respect to the cost (the enrichment functions used in MultiPie and ImageNet dataset follow such correlation).

For each enrichment function correlation, we compared all SB plan generation strategies (as described in Exp 1) on query Q4 using F_1 measure (Figure 12). Since the function with highest $\frac{quality}{cost}$ is always chosen first in SB (FO) strategy, it always outperforms both SB(OO) and SB(RO). However, SB (RO) becomes almost as good as SB(FO), when the correlation is linear (Figure 12(a)), since all enrichment functions have the same $\frac{quality}{cost}$, when the correlation is linear. SB (OO) performs worst as it executes all enrichment functions (including functions with low $\frac{quality}{cost}$), before enriching another tuple.

The correlations of the functions in real datasets are shown in Figure 11 (b). From Figures 8(b), 8(c), and 8(d) of Exp 2, we observe that the quality of answer for BB(DT) improves very aggressively in MultiPie as compared to ImageNet and TweetData. The reason is that the quality of functions used in MultiPie has highest positive curvature in quality-cost graph as compared to ImageNet and TweetData functions.

7 RELATED WORK

The related works on online AQP [29, 40, 90, 94], IMV [4, 56, 67], and recent advances on IQP systems [82–84] were discussed in the introduction §1.Additional related works are discussed here. **Progressive and query-driven approaches** have been studied in several domains: entity resolution [10, 12, 13, 70, 87], crowd-sourced data processing tasks [14, 71, 72], online schema matching [63, 77], and probabilistic databases [32, 78, 79]. However, none of them considered progressive query processing with enrichment. **Systems for supporting ML using databases** (*e.g.*, Apache MADlib [41], Bismarck [37], RIOT [95], SystemML [20, 39], SimSQL [23]) are designed to learn ML models inside or on top of database systems; EDB is related to them, but complementary and focuses on optimally enriching data at query time, using models learnt by these systems. Recent systems [30, 54] address the problem of model selection and optimal plan generation for ML inference queries *without supporting* progressive query execution. **Expensive predicate optimization** problems optimize queries containing expensive predicates [16, 26, 42, 52, 53, 57, 61, 62]. [52, 57, 61] address *optimization of multi-version predicates* using less expensive predicates to filter objects before evaluating expensive predicates. However, [52, 57, 61] considered predicates to be static and deterministic. In contrast, EDB supports both deterministic and non-deterministic enrichment functions/predicates. Offline AQP systems [7, 25, 35, 73] maintain samples beforehand and select the best sample for query execution using an error-latency profiling. Such systems do not consider data enrichment during query processing and also do not consider that underlying data can change due to enrichment as addressed by EDB.

8 CONCLUSION

We proposed EDB, a data management system that supports policy-based enrichment all through the data processing pipeline, at ingestion, periodically based on events, or during query execution. EDB supports mechanisms that exploit contexts to progressively answer queries — initially, approximate results are returned that are improved over time as more relevant data is enriched. EDB is generic and supports multiple attributes that need to be enriched, diverse enrichment functions, and can be layered on different DBMSs. Experimental results on real and synthetic datasets show the efficacy of EDB. While, EDB is implemented using a layered approach, exploring additional optimizations that are possible at the storage layer and modifying query processing layer of DBMS, are interesting future directions for us.

9 APPENDIX

9.1 Discussion on Model Table

This section describes the model table of TAGDB 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 17. 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[41].

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 17: Model table for multi-layered perceptron classifier.

9.2 Parallel Query Execution

The algorithm of query execution as described in §4 considers a single core query execution. TAGDB implementation exploits parallelism in a machine with multi-core CPUs. Although current implementation of parallel query execution uses a simple strategy of query splitting to exploit parallelism, there is a scope for developing efficient parallel algorithms. We describe the query execution strategy using the example query of Figure 2.

The basic idea is to split the original query into multiple smaller queries depending on the number of available cores (referred to as *num_cores*) and execute such queries in parallel. The split queries are referred to as *split_query*[1], *split_query*[2], . . ., *split_query*[*num_cores*]). The original query is split according to the predicates on the fixed attributes.

As a first step, the range selection conditions on fixed attributes are split equally to create the smaller queries. When range conditions exist on multiple fixed attributes, one of the fixed attribute is chosen and the corresponding range is divided equally based on *num_cores*. Note that such ranges are adaptive and are not of equi-width in later epochs. The ranges are updated based on the number of straggler queries (*i.e.*, slow running) at the end of each epoch. If there are no straggler queries, the ranges are kept the same as the previous epoch. *E.g.*, let us consider that the query of Figure 2 is being executed on a machine with 4 CPU cores. EDB divides the range condition on `TweetTime` into four parts and create four split queries as follows:

```
sq1: SELECT ... FROM TweetData WHERE sentiment = '
     positive' AND topic = 'social media' AND
     tweet_time BETWEEN ('16:00', '16:30');
sq2: SELECT ... FROM TweetData WHERE sentiment = '
     positive' AND topic = 'social media' AND
     tweet_time BETWEEN ('16:30', '17:00');
sq3: SELECT ... FROM TweetData WHERE sentiment = '
     positive' AND topic = 'social media' AND
     tweet_time BETWEEN ('17:00', '17:30');
sq4: SELECT ... FROM TweetData WHERE sentiment = '
     positive' AND topic = 'social media' AND
     tweet_time BETWEEN ('17:30', '18:00');
```

Code Listing 1: Queries generated from the original query

In the above queries "..." represent the same attributes of the original query. TAGDB addresses the following fundamental challenges of parallel execution of queries: (*i*) detection of stragglers, (*ii*) mitigation of stragglers, and (*iii*) reconstruction of answers from multiple split queries for the original query.

To address the above challenges, in TAGDB, we have implemented the `pause`, `resume` and `stop` functionalities of queries. The idea is to achieve synchronized execution of different queries by pausing straggler queries and resume queries that finished early to reach the end of the epoch. In the following, we describe the strategies in details.

Straggler Detection. The detection of straggler is performed by calculating a *score* for each of the split queries. The *score* measures the amount of enrichment performed by the query in the current epoch and it is calculated by the ratio of amount of enrichment already completed (*i.e.*, number of tuples in plan table already executed) to the total amount of enrichment assigned in the epoch (*i.e.*, number of tuples in plan table). The score of different queries in an epoch is used to determine the stragglers.

Let us consider that the epoch duration of the queries are 10 seconds. In a certain epoch e_k, suppose that queries $sq1$ and $sq2$

finished their data enrichment phase in 5 seconds and 6 seconds respectively. Similarly, the queries $sq3$ and $sq4$ are expected to finish their data enrichment phases in 12 seconds and 14 seconds respectively. In this epoch, the queries $sq3$ and $q4$ are considered to be the stragglers.

Straggler Mitigation. Traditionally, in parallel computing systems such as MapReduce(MR) systems, when a straggler job is detected, it is automatically replicated to another node [33]. The result of the fastest finished job is used by the MR scheduler and it discards the other unfinished task in the other node and it releases the computing resources back to use. In the presence of a straggler, the mitigation strategy of TAGDB creates a new set of queries instead of continuing with the same set of old queries of the previous epoch. In the above query examples, the mitigation strategy will create the ranges on `TweetTime` attribute using the *score* of the split queries in the last epoch. *E.g.*, if the scores were 1, 0.84, 0.72 and 1 respectively in the previous epoch, then the range of `TweetTime` is split according to the same ratio of progressive score (*i.e.*, $1 : 0.84 : 0.72 : 1$).

This design decision is a departure from traditional parallel computing strategies that distribute the workload of only the straggler jobs. The resulting predicates in split queries are contiguous with each other and EDB do not have to perform complex rewrites with `OR` clause for expressing non-contiguous predicates. The decisions of creating the new set of split queries are taken at the end of each epochs.

Reconstruction of answers. EDB creates a view on top of the IMVs maintained for the split queries using `UNION ALL` command.

Sample query execution. Let us consider the same query of Figure 2, and the split queries of $sq1$, $sq2$, $sq3$, and $sq4$ as shown earlier. Suppose, *epoch_duration* is 10 seconds where in epoch e_k split queries $sq1$ and $sq2$ finished enrichment tasks in 5 and 6 seconds respectively, whereas $sq3$ and $sq4$ are expected to finish in 12 and 14 seconds. EDB continues the execution of split queries $sq1$ and $sq2$ for the remaining duration of epochs (*i.e.*, 5 and 4 seconds respectively) and then stopped using the `stop` command at the end of *epoch_duration*. Similarly, queries $sq3$ and $sq4$ are stopped at the end of *epoch_duration* although their enrichment tasks were not completed. EDB re-splits the queries as discussed earlier in straggler mitigation and start them in the new epoch.

REFERENCES

[1] Please find the codebase at https://github.com/DB-repo/enrichdb.
[2] Apache kafka. https://kafka.apache.org/23/documentation/streams/.
[3] Apache spark. https://databricks.com/spark/about.
[4] Incremental view maintenance development for postgresql. https://github.com/sraoss/pgsql-ivm.
[5] Incremental view maintenance for postgresql. https://wiki.postgresql.org/wiki/Incremental_View_Maintenance.
[6] Tensorflow. https://www.tensorflow.org/.
[7] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42. ACM, 2013.
[8] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154. ACM, 2006.
[9] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.
[10] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 7(11):999–1010, 2014.
[11] Y. Altowim and S. Mehrotra. Parallel progressive approach to entity resolution using mapreduce. In *ICDE*, pages 909–920. IEEE Computer Society, 2017.
[12] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. *PVLDB*, 6(14):1846–1857, 2013.
[13] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3):120–131, 2015.
[14] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD Conference*, pages 241–252. ACM, 2013.
[15] S. Arumugam, F. Xu, R. Jampani, C. Jermaine, L. L. Perez, and P. J. Haas. Mcdb-r: Risk analysis in the database. *Proc. VLDB Endow.*, 3(1–2):782–793, Sept. 2010.
[16] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.
[17] L. Becker, G. Erhart, D. Skiba, and V. Matula. AVAYA: sentiment analysis on twitter with self-training and polarity lexicon expansion. In *SemEval@NAACL-HLT*, pages 333–340. The Association for Computer Linguistics, 2013.
[18] L. Berrada, A. Zisserman, and M. P. Kumar. Smooth loss functions for deep top-k classification. *arXiv preprint arXiv:1802.07595*, 2018.
[19] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.
[20] M. Boehm, M. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. Systemml: Declarative machine learning on spark. *Proc. VLDB Endow.*, 9(13):1425–1436, 2016.
[21] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 1997.
[22] P. Brucker. *Scheduling Algorithms*. Springer Publishing Company, 2010.
[23] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD Conference*, pages 637–648. ACM, 2013.
[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.
[25] S. Chaudhuri, G. Das, and V. R. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2):9, 2007.
[26] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 1999.
[27] J. Chen, C. Xiong, and J. Callan. An empirical study of learning to rank for entity search. In *SIGIR*, pages 737–740. ACM, 2016.
[28] W. Cheng, E. Hüllermeier, and K. J. Dembczynski. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, pages 279–286, 2010.
[29] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, pages 313–328. USENIX Association, 2010.
[30] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627. USENIX Association, 2017.
[31] N. F. F. da Silva, L. F. S. Coletta, and E. R. Hruschka. A survey and comparative study of tweet sentiment analysis via semi-supervised learning. *ACM Comput. Surv.*, 49(1):15:1–15:26, 2016.
[32] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, Oct. 2007.
[33] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
[34] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li. Imagenet: A large-scale hierarchical image database. In *CVPR*, pages 248–255. IEEE Computer Society, 2009.
[35] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD Conference*, pages 679–694. ACM, 2016.
[36] S. Feng, A. Huber, B. Glavic, and O. Kennedy. Uncertainty annotated databases - A lightweight approach for approximating certain answers. In *SIGMOD Conference*, pages 1313–1330. ACM, 2019.
[37] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD Conference*, pages 325–336. ACM, 2012.
[38] J. H. Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.
[39] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242. IEEE Computer Society, 2011.
[40] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.
[41] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or MAD skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.
[42] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 1993.

[43] 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.

[44] R. Hsu, M. Abdel-Mottaleb, and A. K. Jain. Face detection in color images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(5):696–706, 2002.

[45] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: a probabilistic database management system. In *SIGMOD Conference*, pages 1071–1074. ACM, 2009.

[46] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679 – 688, 2006.

[47] T. Imielinski and W. L. Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[48] T. Imielinski, R. van der Meyden, and K. V. Vadaparty. Complexity tailored design: A new design methodology for databases with incomplete information. *J. Comput. Syst. Sci.*, 51(3):405–432, 1995.

[49] P. JACCARD. Etude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[50] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: A monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 687–700, New York, NY, USA, 2008. Association for Computing Machinery.

[51] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. The monte carlo database system: Stochastic analysis close to the data. *ACM Trans. Database Syst.*, 36(3):18:1–18:41, 2011.

[52] M. Joglekar, H. Garcia-Molina, A. G. Parameswaran, and C. Ré. Exploiting correlations for expensive predicate evaluation. In *SIGMOD Conference*, pages 1183–1198. ACM, 2015.

[53] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically optimizing queries over large scale data platforms. In *SIGMOD Conference*, pages 943–954. ACM, 2014.

[54] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending relational query processing with ML inference. In *CIDR*. www.cidrdb.org, 2020.

[55] H. Kimura, S. Madden, and S. B. Zdonik. UPI: A primary index for uncertain databases. *Proc. VLDB Endow.*, 3(1):630–637, 2010.

[56] 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.

[57] I. Lazaridis and S. Mehrotra. Optimization of multi-version expensive predicates. SIGMOD, 2007.

[58] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[59] J. Li and A. Deshpande. Consensus answers for queries over probabilistic databases. In *PODS*, pages 259–268. ACM, 2009.

[60] Y. Lin, D. Jian, R. Yus, G. Bouloukakis, A. Chio, S. Mehrotra, and N. Venkatasubramanian. Locater: Cleaning wifi connectivity datasets for semantic localization.

[61] Y. Lu, A. Chowdhery, S. Kandula, and S. Chaudhuri. Accelerating machine learning inference with probabilistic predicates. In *SIGMOD Conference*, pages 1493–1508. ACM, 2018.

[62] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *SIGMOD Conference*, pages 659–670. ACM, 2004.

[63] R. McCann, W. Shen, and A. Doan. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, pages 110–119. IEEE Computer Society, 2008.

[64] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*. www.cidrdb.org, 2013.

[65] K. Mikolajczyk, C. Schmid, and A. Zisserman. Human detection based on a probabilistic assembly of robust part detectors. In *ECCV (1)*, volume 3021 of *Lecture Notes in Computer Science*, pages 69–82. Springer, 2004.

[66] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[67] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264. ACM, 2014.

[68] R. Nuray-Turan, D. V. Kalashnikov, S. Mehrotra, and Y. Yu. Attribute and object selection queries on objects with probabilistic attributes. *ACM Trans. Database Syst.*, 37(1):3:1–3:41, 2012.

[69] R. Olfati-Saber and J. S. Shamma. Consensus filters for sensor networks and distributed sensor fusion. In *CDC*, pages 6698–6703. IEEE Computer Society, 2005.

[70] T. Papenbrock, A. Heise, and F. Naumann. Progressive duplicate detection. *IEEE Trans. Knowl. Data Eng.*, 27(5):1316–1329, 2015.

[71] A. G. Parameswaran, S. P. Boyd, H. Garcia-Molina, A. Gupta, N. Polyzotis, and J. Widom. Optimal crowd-powered rating and filtering algorithms. *Proc. VLDB Endow.*, 7(9):685–696, 2014.

[72] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD Conference*, pages 361–372. ACM, 2012.

[73] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *SIGMOD Conference*, pages 1461–1476. ACM, 2018.

[74] 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.

[75] J. Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

[76] D. M. Powers. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *J. Mach. Learn. Technol*, 2:2229–3981, 01 2011.

[77] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[78] C. Ré, N. N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, pages 886–895. IEEE Computer Society, 2007.

[79] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, pages 596–605. IEEE Computer Society, 2007.

[80] T. Sim, S. Baker, and M. Bsat. The CMU pose, illumination, and expression (PIE) database. In *FGR*, pages 53–58. IEEE Computer Society, 2002.

[81] J. A. K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, 1999.

[82] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, 2019.

[83] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb in action: Resource-efficient query execution by exploiting time slackness. *Proc. VLDB Endow.*, 13(12):2937–2940, 2020.

[84] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In *SIGMOD Conference*, pages 1241–1256. ACM, 2020.

[85] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD Conference*, pages 147–156. ACM, 2014.

[86] X. Wang and M. J. Carey. An IDEA: an ingestion framework for data enrichment in asterixdb. *Proc. VLDB Endow.*, 12(11):1485–1498, 2019.

[87] S. E. Whang, D. Marmaros, and H. Garcia-Molina. Pay-as-you-go entity resolution. *IEEE Trans. Knowl. Data Eng.*, 25(5):1111–1124, 2013.

[88] C. J. Willmott and K. Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[89] D. H. Wolpert. Stacked generalization. *Neural Networks*, 1992.

[90] S. Wu, B. C. Ooi, and K. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD Conference*, pages 651–662. ACM, 2010.

[91] J. Xu, D. V. Kalashnikov, and S. Mehrotra. Query aware determinization of uncertain objects. *IEEE Trans. Knowl. Data Eng.*, 27(1):207–221, 2015.

[92] B. Zadrozny and C. Elkan. Transforming classifier scores into accurate multiclass probability estimates. In *KDD*, pages 694–699. ACM, 2002.

[93] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.

[94] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: generalized on-line aggregation for interactive analysis on big data. In *SIGMOD Conference*, pages 913–918. ACM, 2015.

[95] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with RIOT. In *ICDE*, pages 1157–1160. IEEE Computer Society, 2010.