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

ABSTRACT

This paper proposes TAGDB,[1] a database system designed to support application domains that require incoming data to be enriched prior to being used. Unlike several recent systems (in academia and industry) that have explored ways to optimize enrichment at ingestion, TAGDB integrates enrichment all through the data processing pipeline — at ingestion, triggered based on events, and progressively during query processing. TAGDB is most useful when enrichment is expensive (*e.g.*, requiring ML classifiers to be run). It is based on the premise that enriching data in its entirety at ingestion can be wasteful (if applications do not require all data to be fully enriched), may result in unacceptable latency (if the speed of data arrival is higher than can be enriched), or not be feasible (if enrichment functions are learned and incorporated into the system at a later time after ingestion during analysis). TAGDB has been implemented using a layered approach on top of PostgreSQL, though it can easily be layered on other databases. This paper describes the architecture, design choices, and implementation of TAGDB, as well as, evaluates its performance focusing on TAGDB's support for enrichment during progressive query answering.

1 INTRODUCTION

TAGDB is a system that transparently supports the enrichment of (possibly) raw uninterpreted data (*e.g.*, text, images, and sensor data) into semantically meaningful data that can be used to support analytical applications. Examples of such transformations include entity linking [29], sentiment analysis of tweets [18, 34], semantic interpretation of sensor data [71], and data transformations using machine learning (ML) algorithms to interpret/classify images [49, 68] in different categories. Traditionally, enrichment is performed as a 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. Additionally, such enrichment is wasteful, when queries have varying needs requiring either data to be enriched partially and/or requiring only part of data to be enriched.

Recently, several industrial systems (*e.g.*, Apache Storm [92], Spark Streaming [100], Flink [25], Kafka Stream [2]) have explored data enrichment at scale during ingestion, including techniques [93] to batch input data to reduce ingestion overhead. Such systems target scenarios where simple functions (*e.g.*, that do not add significant latency) can be executed at ingestion, even when data arrives very fast; however, they do not scale to situations where online data analytics requires execution of a suite of computationally-expensive functions. We illustrate the challenge from our experience in building a real-life application that serves as a motivation to develop TAGDB.

Motivating Example. Consider a campus with a large number of buildings (≈100), users (≈10K), WiFi access points (several thousand), Bluetooth beacons (ten thousand), and cameras (several thousand), and a location monitoring application that tracks users to support location-based services.[2] Such sensors produce ≈100K WiFi and Bluetooth events and ≈10K camera images per second. While coarse-level localization based on Bluetooth beacon and/or WiFi access point data can be performed as data streams in (such code takes ≈10ms/event on a server with 64 core Intel Xeon CPU E5-4640 with 2.40GHz clock speed and 128GB memory), performing fine-grained localization (*e.g.*, using face detection on camera data and/or using expensive ML techniques on multi-sensor data) can take ≈1s/event on the same server, making it infeasible to perform at ingestion. The challenge here is not of fast ingestion — stream processing systems ([2, 25, 92, 100]) can handle such data rates, instead, the complexity arises from having to execute expensive enrichment functions on data as it arrives (*e.g.*, fully processing all events generated in one second of the campus sensing infrastructure takes about 5 hours). A detailed case study is available in 8.1.

The example above emphasizes the smart building use-case of TAGDB. Similar scenarios exist in other application contexts as well, such as tweet or web traffic monitoring to track events or possible network attacks in real-time over social media using enrichment functions of topic detection, entity linking, sentiment analysis, and network surveillance platforms.

Key Technologies. To effectively support data enrichment for real-time analytics, TAGDB supports two key technologies:

(1) *Policy-based enrichment mechanism:* that supports data enrichment based on policies at all stages of data processing, *i.e.*, ingestion, intermittently based on workload characteristics, and at query time. Enriching data at different stages offer different benefits: when performed at ingestion, it speeds up query response time that would have increased, if enrichment was delayed until query execution. In contrast, enriching at query time alleviates load at ingest,[3] and limits redundant enrichment of data that is not part of any query. Intermittent enrichment attempts to alleviate the load at ingestion, while simultaneously reducing query latency.

(2) *Progressive query processing mechanism:* that provides query answers with gradually improving quality based on "just-in-time" enrichment performed on the objects during query execution. Progressive query processing allows newly ingested data (that has not yet been enriched) to be available for analysis right away, without having to wait for enrichment on data to complete.

To support data enrichment at any phase of data processing, TAGDB, explicitly, represents the state of enrichment of objects (*i.e.*, which part of data has been enriched using which functions), a policy specification mechanism that captures what enrichment tasks should be performed on the data when the data is ingested or during offline pre-processing. Such policies could be either pre-specified by users before the arrival of the data (as supported in the current implementation of TAGDB) and/or self-learned using previous data ingestion and query workload (ongoing work).

[1] We name it as TAGDB as the enrichment functions are used to tag data and the queries are executed progressively in a *think-act-give* (*i.e.*, *tag*) cycle.

[2] Location tracking can raise significant privacy concerns but the issue is orthogonal to this paper. In our deployment (built using TAGDB), users opt-in to the system and data access is based on user policies.

[3] Such a benefit has been the driving force behind *data lakes* [44, 91] that ingest data into storage in its raw form and support structuring of the data on demand when queried.

In TAGDB, users specify data types for storing data from various data sources, mark certain attributes (that require enrichment) as *derived attributes*. Users can dynamically add/remove/update functions that can be used to enrich such attributes. In general, a derived attribute is associated with multiple enrichment functions. The policy specification mechanism is used to represent the enrichment policies for derived attributes in a relation. A policy specifies how data of a particular data type needs to be enriched (using which functions) and at which stage of data processing pipeline (*i.e.*, at ingestion, periodically, or at query processing time). Progressive query processing is achieved using an iterative approach, wherein execution time is divided into multiple smaller time intervals, called *epochs* (defined in §3). Each epoch, itself, consists of three components: *plan generation* (*i.e.*, *think*) that selects an enrichment plan consisting of which tuple and which attribute to enrich using which functions, a *plan execution* (*i.e.*, *act*) that executes the enrichment plan, and *query evaluation* (*i.e.*, *give*) that incrementally produces answers from partially enriched data at the end of the epoch.

Relationship to Online Approximate Query Processing (AQP) Systems. Progressive query answering in TAGDB is related to online AQP systems [32, 45, 96, 101] that progressively improve aggregate query results over time.[4] In online AQP, iterative sampling [45] is used to improve the quality of answer for aggregation queries over very large datasets. Several sampling mechanisms have been explored, *e.g.*, uniform sampling [27, 32], stratified sampling [9], and variational sub-sampling [76]. The key design considerations of an AQP system include efficient sample selection, maintenance, and query time sample size determination ([8, 9, 26]).

In contrast, TAGDB supports progressive query processing, not as much to deal with large data volume, but to overcome the high cost of enrichment functions during query execution. The key design considerations of TAGDB includes: selection of the right set of tuples and functions to enrich in the context of the query, efficient execution of the enrichment functions, efficient computation of the incremental results, and efficient state management of tuples.

Layered Implementation. TAGDB is implemented using a ***layered approach*** on top of PostgreSQL. The layering consists of appropriate translation of data types to PostgreSQL tables and association of enrichment functions with user-defined functions (UDFs). TAGDB stores enrichment policies, enrichment state of tuples, and the metadata in the form of PostgreSQL tables that are used during ingestion and query processing. An insert command in TAGDB is transformed into a PostgreSQL insert command based on the policy table. A TAGDB query is wrapped in a stored procedure, entitled TagDB *executor* (discussed in detail in §4.5) that internally executes appropriate SQL queries on top of PostgreSQL tables during multiple epochs. The execution of the enrichment functions (*i.e.*, UDFs) in these rewritten queries is orchestrated by a special UDF entitled *driver UDF* that acts as a meta-UDF, executing the UDFs that are passed to it as arguments (see §4.5.3).

The layered implementation of TAGDB, while limits the set of design options compared to building a specialized system built from scratch, offers several advantages: First, a layered implementation of TAGDB is largely platform-agnostic – it exploits common features available in a large number of modern databases, viz., indexes, stored procedures, and materialized views. Even though current implementation is built on top of PostgreSQL, adapting TAGDB to run on top of other databases require only minor changes to the storage, enrichment, and query modules of TAGDB. Thus, a layered implementation allows organizations/implementations already vested in specific database technology to potentially benefit from progressive enrichment and query answering of TAGDB without having to migrate completely to a new system. Another advantage is that it allows comparison between different query plan generation strategies and query semantics, relatively easily compared to native implementation (§5.1 will compare multiple such strategies and show that just with a layered implementation we achieve significant progressiveness during query execution). Note that layered implementation has often been a preferred route for several new technologies [7, 40, 76, 77] for reasons similar to those listed above.

Outline of the paper. §2 describes the data model. §3 describes the query model. §4 describes TAGDB architecture, implementations, and details of query processing. §5 provides experimental evaluation. In Appendix §8.1, we present a complete case study that serves the motivation behind TAGDB.

2 TAGDB DATA MODEL

TAGDB 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.[5]

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

The value of a derived attribute is determined using one or more ***enrichment functions*** associated with it. We refer to the value of derived attributes as ***tags*** that (as will become clear in §3.1) are internally represented as a set of possible values that the derived attribute can take based on the output of 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 [89] that outputs a *yes/no* answer for predicting a label), or multiple values (*i.e.*, multi-label classifier that outputs top-k predicted labels [19]), or a probability distribution (*i.e.*, probabilistic classifier that outputs a probability distribution over predicted labels [31]).

In general, we consider an enrichment function as a mapping from a value of the derived attribute to a probability distribution over the domain of the derived attribute.[6] In the example above, the enrichment functions for `sentiment` (and `topic`) attribute will return distribution over three (and forty) values. Each enrichment function is associated with a notion of *cost* and *quality*, where *cost* refers to the average execution time of the enrichment function on a single tuple and *quality* 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.

[4] AQP systems can be classified into: offline systems ([9, 26, 39, 76]) and online systems ([32, 45, 96, 101]) based on the way sample of tuples are selected. Online aggregation systems are more related to the progressive query processing used in TAGDB.

[5] Keyword `StateCutoff` used in `CREATE TABLE` statements will be explained in §4.4. 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 [62] pre-extracted from tweets.

[6] Probability values are undefined/NULL, for non-probabilistic classifiers that output one/more labels.

The enrichment function could be either selected from a set of functions included in TAGDB (*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, TAGDB 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 [46]. 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 thte `training procedure`. Likewise, there could be other functions based on random forest, decision tree (DT), or naïve Bayes classification algorithm to compute `sentiment` values. Similarly, we can have functions to compute the `topic` (not shown in the example). TAGDB supports a variety of function implementations apart from the one provided in MADlib.

```
CALL tagdb.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, TAGDB splits the training data table into a training and validation dataset).[7] The *quality* of the function is computed by executing it on the validation dataset and measuring the *area under the curve* (AUC score) [22] (using the model learned on the training dataset). 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 a ***function-family*** of $\mathcal{A}_i$. The function-family of $\mathcal{A}_i$ is created for $\mathcal{A}_i$ using `assign` function. We use ***calligraphic font*** for ***derived attributes***.

```
CALL tagdb.assign(
 'tweets_sentiment_family', 'TweetData', 'sentiment',[
 ['mlp_classifier', 'sentiment_mlp_model',0.10,0.86],
 ['dt_classifier', 'sentiment_dt_model', 0.06, 0.7]]);
```

Above, `assign` function creates a function-family for `sentiment` derived attribute,[8] and contains functions *mlp_classifier*() and *dt_classifier*(). For example, *mlp_classifier*() uses `sentiment_mlp_model` for sentiment prediction, and it has *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***. TAGDB currently supports `weighted average` (default), `majority voting`, and `majority average` based combiner functions. Future version of TAGDB will support other complex combiner functions, *e.g.*, stacking [95] and boosting-based [42] combiners.

[7] TAGDB can use a sample from the same training dataset used by `training procedure` provided by the user to learn models associated with the enrichment functions as validation dataset.

[8] A function-family may have a name, *e.g.*, `tweets_sentiment_family` in the above example.

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 non-zero 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 §4.4.)

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 $state(t_k) = \langle state(t_k.\mathcal{A}_p) || state(t_k.\mathcal{A}_q) || state(t_k.\mathcal{A}_r) \rangle$. TAGDB stores the tuple states in a different table (instead of in-lining with t_k). The rationale behind this design choice is explained later in §4.4. The state of a TAGDB 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).

3 TAGDB QUERY MODEL

TAGDB 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 the user with the query. 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 and yet be able to get results of high quality. In addition, epoch-based execution empowers TAGDB to adapt execution at epoch boundaries by choosing enrichment functions to execute.

To support the above execution strategy, TAGDB determines the value of the derived attributes based on its current state. The determinized attribute value could change across epochs as more enrichment functions execute, resulting in progressively improved answers. Below, we discuss: (*i*) how the attribute values are determined given their state (§3.1), semantics of query results over partially enriched data (§3.2), and the strategy of incrementally computing query results across epochs (§3.3).

3.1 Determinization Function

In TAGDB, the value of an attribute $t_k.A_i$ 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 either a set of one or more possible values of $\mathcal{A}_i$. It may also return a NULL value, which represents the situation when the state of the attribute, based on enrichment functions executed so far, does not provide enough evidence to either assign or eliminate any domain value as a possible value for $\mathcal{A}_i$.

The choice of determinization function depends on the output type of the underlying enrichment functions and the combiner function.

tid	UserID	TweetObject	feature	location	TweetTime	**topic**	**sentiment**
t_1	John	Uploading pics on Facebook	[0, 0.25, 0, 0, 0, 0,0.41,0,0.46]	US	2020-02-20 16:08:00	social media, entertainment	positive, neutral
t_2	Mark	Feeling fine & listening Semisonic.	[0.2,0,0.54,0.1,0.16, 0,0.58,0,0,0.63]	US	2020-02-20 16:48:00	entertainment, art	NULL
t_3	Richard	Sad about Iran's political situation.	[0, 0.67, 0, 0.58, 0, 0, 0,0,0,0,0.74]	UK	2020-02-21 11:48:00	politics, geography	neutral, negative

Table 1: `TweetData` table in TAGDB. Derived attributes are `topic` and `sentiment` and the values represent their determinized representation.

tid	topic state-output	sentiment state-output
t_1	social media:0.54, entertainment:0.38, sport:0.08	positive:0.52, neutral:0.44, negative:0.04
t_2	entertainment: 0.45, art: 0.35, politics: 0.1	positive:0.33, neutral: 0.34, negative: 0.33
t_3	politics:0.58, geography: 0.34, entertainment:0.08	positive:0.1, neutral: 0.42, negative: 0.48

Table 2: State of `TweetData`. The threshold for topic is set as 0.3 and for sentiment as 0.4.

For instance, if enrichment functions correspond to non-probabilistic classifiers such as [19, 89] (that output a single or top-k class labels) and a combiner function (*e.g.*, majority voting) that takes such an input and outputs a list of possible labels are selected, then the determinization function simply returns the output of the combiner function. If probabilistic classifiers [31] (that output a probability distribution over possible domain values) are used as enrichment functions and a combiner function (*e.g.*, weighted average) outputs a probability distribution over the domain of attribute, then any function that maps such a probability distribution into a set of possible choices can serve as a determinization function. Previous literature has explored threshold-based and quality-maximization based determinization [64, 70, 97]. TAGDB can use any of these mechanisms as a determinization function. Since the choice of determinization function is not a contribution of this paper, we choose a threshold based determinization function in our experiments.

The concept of determinization naturally extends to a tuple and a relation. 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 3.1. To illustrate the determinization process, consider a relation `TweetData` with two derived attributes `topic` and `sentiment` (see Table 1),[9] and assume state-output of the two derived attributes, shown in Table 2. We will explain in §4.4 how such a state table is created. Based on Table 2 and threshold-based determinization strategy, where thresholds for `topic` and `sentiment` are set as 0.3 and 0.4 respectively, 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.■

3.2 Query Semantics

Since the determinization of a tuple can result in either a set of values or NULL, the evaluation logic of different types of conditions needs to be defined. The condition evaluation in TAGDB extends the traditional three-valued logic used in relational operators, *i.e.*, with truth values of T, F, and U into a four-valued logic: **true (T), false (F), possible (P), and unknown (U)**. Here, P represents that the condition is **possibly true** based on the current state of enrichment, whereas U (as in traditional setting) represents that the truth value is **unknown**, given the current level of enrichment. Similar to SQL, TAGDB does not return tuples that evaluate to unknown in the answer set. However, as will become clear in §4.5, the tuples evaluating to **possible** may or may not be returned. *E.g.*, inclusion of such tuples in the answer could be based on maximization of progressive score of the query, as defined in §3.3. We next discuss how TAGDB assigns truth values to predicates/expressions.

- ***Simple Predicates.*** Consider an expression $\mathcal{A}_i$ `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 the output of $DET(state(t_k.\mathcal{A}_i))$ is NULL, then, as in traditional SQL, the expression evaluates to U. If the output $DET(state(t_k.\mathcal{A}_i))$ is a singleton (say S) and for an element $x \in S$ such that x `op` a_m holds, then the expression evaluates to T; otherwise, it evaluates to F. If the output of $DET(state(t_k.\mathcal{A}_i))$ is a multi-valued set (say S) and for an element $x \in S$ such that x `op` a_m holds, then it is possible that the tuple t_k satisfies the expression, and hence, it evaluates to P. However, if there does not exist any element $x \notin S$, then the expression cannot be true, and hence, evaluates to F.

  Consider an expression $\mathcal{A}_i$ `op` $\mathcal{A}_j$, where $\mathcal{A}_i$ and $\mathcal{A}_j$ are two derived attributes of (possibly) different relations, and `op` is a comparison operator. If the output of $DET(state(t_k.\mathcal{A}_i))$ or $DET(state(t_l.\mathcal{A}_j))$ is NULL, then the 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 sets and for elements $x \in DET(state(t_k.\mathcal{A}_i))$ and $y \in DET(state(t_l.\mathcal{A}_j))$ 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 $x \in DET(state(t_k.\mathcal{A}_i))$ and $y \in DET(state(t_l.\mathcal{A}_j))$, such that x `op` y holds, then the predicate evaluates to P; otherwise, F.
- ***Complex Predicates.*** Complex predicates can be formed using multiple comparison conditions connected using Boolean connective operators (AND ($\wedge$), OR ($\vee$), and NOT ($\neg$)). Table 3 shows the truth table for such logical operators. Table 3 only shows entries when one of the two expressions evaluates to P. When both expressions evaluate to either T, F, or U, TAGDB 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

Table 3: Truth table for evaluating conditions.

Based on the definition of determinization function and the predicate evaluation logic described above, the query semantic for TAGDB is defined as follows:

$$q(R_1,R_2,...,R_n) = q'(DET(R_1),DET(R_2),...,DET(R_n))$$

where $q(R_1,R_2,...R_n)$ is a query on relations $R_1,...,R_n$, $DET(R_i)$ 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 the predicate evaluation logic explained above.

Example 3.2. Consider a query on `TweetData` (Table 1) that retrieves all tweets on '`topic = social media`', '`sentiment = positive`, and posted between "2020-02-20" and "2020-02-21" (Figure 1). The query is executed on the determinized representation of tuples as described in Example 3.1. The result is shown in Table 4. ■

```
SELECT TweetObject, location, TweetTime, topic,sentiment FROM
    TweetData WHERE sentiment = 'positive' AND topic = 'social
    media' AND TweetTime BETWEEN ('2020-02-20', '2020-02-21');
```

Figure 1: An example query on `TweetData`.

TweetObject	location	TweetTime	topic	sentiment
Uploading pics on Facebook	US	2020-02-20 16:08:00	social media, entertainment	positive, neutral

Table 4: Result of the query shown in Figure 1.

[9] The `feature` attribute contains term frequency-inverse document frequency (tf-idf) vector [62] pre-extracted from tweets.

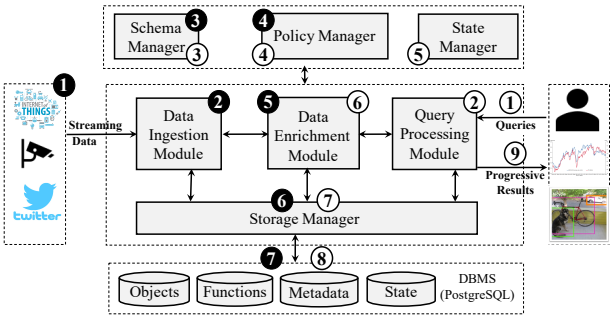


Figure 2: TAGDB Architecture (data ingestion flow is in black-circles, and query execution flow is in white-circles).

3.3 Progressive Query Execution

Since TAGDB produces query answers in epochs, TAGDB supports a mechanism to refine previously produced query answers. This is achieved by retracting or adding tuples for set-based results or by improving the confidence intervals [45] 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 $\Delta(q,e_k)$ is the set of tuples added to the query answer of epoch e_{k-1} and $\nabla(q,e_k)$ is the set of tuples removed from the query answer of 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 and the corresponding confidence-interval. In epoch e_k, if either the aggregated value or the confidence interval 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/confidence interval is added.

Progressive Score. Since in TAGDB, users may stop query evaluation at any instance of time, performing enrichments that impact the answer quality as early as possible is desirable. TAGDB's effectiveness is measured using the following progressive score (similar to other progressive approaches [11, 12, 73]):

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

where $E = \{e_1, e_2, ..., e_k\}$ is a set of epochs, $W(e_i) \in [0,1]$ is the weight allotted to the epoch e_i, $W(e_i) > W(e_{i-1})$), Q is the quality of answers, and $[Q(Ans(q, e_i)) - Q(Ans(q, e_{i-1}))]$ is the improvement in the quality of answers occurred in epoch e_i. In Equation 2, the quality Q of a set-based query answer can be measured using set-based quality metrics: precision, recall, F_α-measure [80], or Jaccard similarity coefficient [52]. The quality Q of an aggregation query can be measured using root-mean-square error [51] or by adding the inverse of half-interval lengths[10] of *GROUP BY* keys. Given a query, the maximum number of epochs, a quality metric for answers, and a set of weights assigned to epochs, the goal of TAGDB is to achieve the maximum progressive score for the query.

Quality guarantees. In systems that offer approximate results, it is desirable to associate quality guarantees with the output. For instance, AQP systems that use progressive sampling to refine results of aggregate queries, estimate the error bounds based on processed sample size [9, 45, 76, 82]. In TAGDB, answers (that could be aggregates or set-based results) are progressively refined based on further enrichment of data. Quality guarantees of the results depend upon the state of enrichment, as well as, the choice of the determinization function used. Prior work (*e.g.*, [64, 70]) has explored ways to convert probabilistic datasets (where object values are probabilistic distributions) into determinized representations that optimize a given metric such as F_α-measure or Jaccard similarity. [70] has also considered quality guarantees in the context of selection queries over determinized representation. Exploring quality guarantees on TAGDB results is an interesting direction of future exploration but is outside the scope of this work which focuses on the efficient implementation of enrichment on top of a database system.

[10] The half-interval length refers to the interval value present in the confidence-interval of an aggregation result [45]. For example, an aggregation result with a value of 112, with a confidence of 95%, and an interval of ±2.8, will have the half-interval length of 2.8.

RelationName	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 5: Enrichment policies.

4 TAGDB IMPLEMENTATION

This section describes the layered implementation of TAGDB on top of PostgreSQL, focusing on query processing. We, first, present the overview of TAGDB architecture (§4.1), and then, focus on three main modules: schema manager (§4.3), state manager (§4.4), and query processing module (§4.5) that together implement progressive query execution. Note that in the layered implementation, TAGDB tables and queries are mapped to corresponding PostgreSQL tables and queries. To avoid confusion, we will use the term **TAGDB table (or relation)** and **TAGDB query** to distinguish it from PostgreSQL tables and PostgreSQL queries to which they are mapped.

4.1 Architecture Overview

Figure 2 presents TAGDB architecture highlighting the key components discussed below.

Schema manager (§4.3): stores and manages metadata such as data types, attribute types, function-families, and TAGDB tables. Schema manager provides several APIs for other components to fetch metadata that are used during data ingestion and query processing.

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,[11] and the data pipeline stage at which the policy is applicable (ingestion time, in the background after insertion or query time (which is the default)). Table 5 shows example policies on `TweetData` relation, where, *e.g.*, the first policy states that all tuples of `TweetData` with `TweetText` containing the keyword 'covid' will be enriched using *mlp_classifier*() function at ingestion time. The policy manager provides APIs to insert/delete/update policies.[12]

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. Figure 2 shows the data ingestion workflow using black-filled circles. Data from various data sources (❶) reach the data ingestion module (❷). Given the type of incoming tuples

[11] 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.

[12] 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.

TableName	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 6: `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 7: `FunctionFamily`.

FID	FunctionName	ModelTableName	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 8: `FunctionMetadata`, storing the metadata of enrichment functions.

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

Table 9: `DecisionTable`.

(*e.g.*, tweets, data from WiFi access points, or images from cameras), data ingestion module interacts with schema manager (❸) and policy manager (❹) to check the enrichment policies applicable to them. If an enrichment policy applies to a tuple, the data enrichment module (❺) executes enrichment functions on the tuple depending on the policy. Finally, the tuples and the enrichment function outputs are sent to the storage manager (❻) and are stored in PostgreSQL tables (❼). During ingestion, data is buffered when the *enrichment throughput* cannot sustain the data arrival rate (*i.e.*, during bursty data arrival).

State manager (§4.4): provides APIs for efficient access/update of the state (as specified in §2) of tuples to other modules. These states are used to generate an enrichment plan during query execution.

Storage manager: stores TAGDB 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.

Query processing module (§4.5): produces answers to TAGDB queries in an iterative manner, by performing data enrichment (in each epoch) that achieves progressive quality improvement of results. It provides APIs: (*i*) to start/pause/stop a TAGDB query by the user at any instance of time or after a certain number of epochs (passed as query parameter), (*ii*) to fetch query results progressively, and (*iii*) to retrieve the query and enrichment plans via `EXPLAIN` APIs for the next epoch of the query.

Figure 2 shows the query processing workflow using hollow circles. A user query (①) to TAGDB reaches the query processing module (②) of TAGDB. This module communicates with schema manager (③) and policy manager (④) to determine enrichment policies that were already executed on tuples at ingestion time and/or intermittently. Query processing module then progressively (in epochs) generates and executes an enrichment plan involving enrichment functions not executed earlier. During enrichment, the enrichment functions are executed on tuples with the help of state manager (⑤) and data enrichment module (⑥). Enrichment functions update the state of tuples (using state manager APIs), and produce enriched tuples that are added/updated using storage manager APIs (⑦) in PostgreSQL tables (⑨). TAGDB query is evaluated on (partially) enriched tuples by transforming it to a PostgreSQL query and the answers are sent to the user (⑨). Above tasks ②-⑨ are iteratively performed, *i.e.*, in each epoch.

4.2 User Interfaces

A user interacts with TAGDB 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 TAGDB 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.

TAGDB web application. TAGDB 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, TAGDB 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 [45]. The only difference is that a `GROUP BY` key can be added or removed in a later epoch.

4.3 Schema Manager

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

Metadata of TAGDB relations. `RelationMetadata` (see Table 6) stores the name of each TAGDB relation (in `TableName` column), the names of attributes (in `attribute` column), its type — derived/fixed (in `type` column), the domain size in case of derived attributes (in `NumLabel` column), and a column `StateCutoff` — the purpose of that will become clear in §4.4.

Metadata of enrichment functions: is stored in `FunctionMetadata` table (see Table 8 as an example). TAGDB supports three types of functions: (*i*) functions implemented in Python, using standard ML libraries, *e.g.*, scikit-learn [78], mlextend [5], and TensorFlow [6], (*ii*) analytics library of Apache MADlib [46],

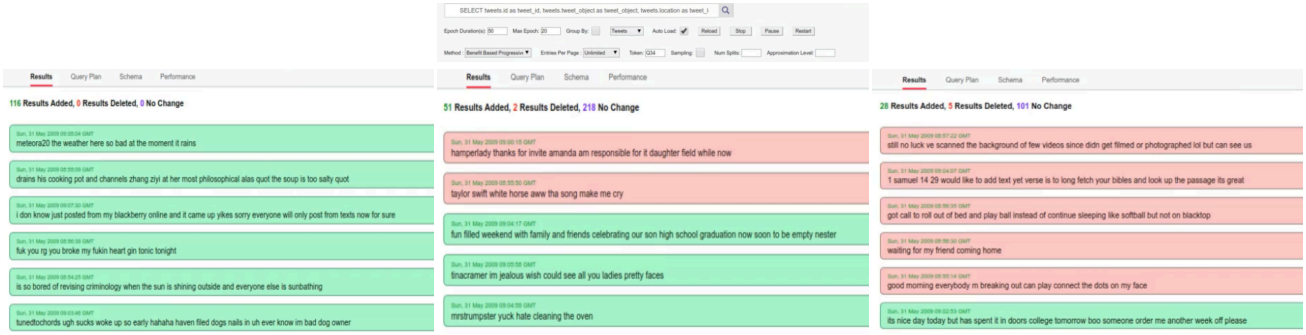


Figure 3: Query Execution in TAGDB for the query of Figure 1. The top figure shows the interface for submitting a query to TAGDB. The bottom figures show the query results at the end of three epochs during query execution. Green box shows newly added tuples and red box shows deleted tuples from previous epoch. Green and red boxed tuples are shown at the the top of result set.

where user can train ML algorithms using SQL, and (*iii*) algorithms implemented in C++/Java language and hosted using a REST API.

`FunctionMetadata` table stores: the column names of TAGDB tables that contain the feature vector (in `InputColumn`), derived attribute name (in `OutputColumn`), the parameters (`params`), the average execution time of function (`cost`), and the quality of the function (`quality`). Some of these functions require a training model. If a model is trained using TAGDB `training procedure`, the model is stored in a `model_table`. For such functions, `FunctionMetadata` table stores `model_table` name (in `ModelTableName` column). For example, in Table 8, function f_1, *i.e.*, *mlp_classifier*(), for sentiment detection on tweets, uses a trained model that is stored in a model table, named sentiment_mlp_model. The representation of the model parameters (*e.g.*, activation function, number of layers in a multi-layer perceptron) in TAGDB's `model_table` follows a similar format as represented in MADlib (`model_table` is discussed in detail in Appendix §8.2).

Function-families: are stored in `FunctionFamily` table (see Table 7) with columns: `FamilyID` (stores function-families' identifiers), `FID` (stores function identities), `AttributeName` (stores the derived attributes' names), and `RelationName` (stores relations' names having derived attributes).[13]

Decision table. Since the 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 TAGDB, 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 by executing the function.

While users can specify customized `DecisionTables` that differ based on the mapping functions used to guide the enrichment process in TAGDB, by default, TAGDB uses a mapping function that — given a state of the attribute — determines the reduction in entropy [48][14] 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.

[13] A function, enhancing different attribute tags, may belong to different function-families.

[14] For a probabilistic distribution over a set of domain values $D_1,...,D_n$ with probabilities $p_1,...,p_n$, entropy is measured as: $\sum_{i=1}^{n} -p_i \log p_i$.

tid	Topic StateBitMap	TopicStateOutput	Sentiment StateBitMap	SentimentStateOutput
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 10: `TweetDataState` table (created for tuples in `TweetData` table).

Table 9 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 $(\langle 1,0,0\rangle, (0\text{-}0.25))$ 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 10) 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 9, since entropy of t_1 is in the range (0.25-0.5), TAGDB determines the next best function to execute on t_1 is f_3 and its benefit as 0.2.

Learning of mapping function. TAGDB 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 §3). For each tuple in the validation dataset, TAGDB 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 9). 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 TAGDB supports the learning of the above map based on entropy, users can provide their own custom maps to guide the selection of functions that TAGDB uses for further enrichment. Alternatively, TAGDB supports a sampling-based method to execute enrichment that does not depend on the decision table as discussed in §4.5.2.

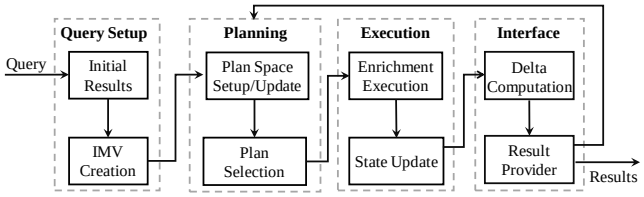


Figure 4: Different stages of query processing implemented inside *executor* procedure.

4.4 State Manager

State manager stores the enrichment states of tuples of each relation in a `State` table (see Table 10) that is created at the time of creation of TAGDB tables. For each derived attribute $\mathcal{A}_i$, the `State` table stores the state-bitmap and the 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 the `TopicStateOutput` column stores the corresponding outputs.

TAGDB stores the state of tuples as a separate table instead of in-lining with the TAGDB table, since the size of the `State` table is usually much smaller compared to the TAGDB table (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 §4.5.2).

State Cutoff Representation. `StateOutput` column implemented naively can be large depending on the domain size of the derived attribute. For example, if the domain size of `topic` is 40 and there are 3 enrichment functions, then the `TopicStateOutput` column (see Table 10) may contain 120 different values in each row. Thus, large domain size may incur storage overhead and read/write cost of such states.

To address this problem, TAGDB uses a compressed representation to store `StateOutput` of tuples for attributes with large domain sizes. For such attributes, TAGDB 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 number 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 by Kimura et al. [58] 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.

4.5 Query Processing Module

Query processing in TAGDB consists of four main steps (see Figure 4): query setup (§4.5.1), planning (§4.5.2), execution (§4.5.3), and interface update (§4.5.4). Query setup step is executed only once, *i.e.*, in the first epoch, whereas the remaining steps are executed iteratively once in each epoch. Note that the ***first epoch is a special epoch*** in which TAGDB sets up the tables used during the enrichment plan generation phase (as will be clear in §4.5.2). A TAGDB query is wrapped in a stored procedure, entitled TagDB *executor* that internally executes all the above-mentioned steps. The pseudo-code for TagDB *executor* is shown in Algorithm 1.

Algorithm 1: Pseudo-code of *executor* stored procedure.

Inputs: $\langle q, \delta \rangle$. // q: a query on relations $R_1,...,R_n$, δ: epoch duration.
Outputs: Query result of q stored in incremental materialized view *IMV*.
Variables: κ: epoch.

```
1 Function ExecuteQuery(q) begin
2     κ ← 1
3     while (!AllEnriched(q) ∧ !QueryTerminated(q)) do
4         if κ = 1 then ⟨q′,IMV⟩ ← QuerySetup(q) // rewrites q to
              q′, creates IMV.
5         PlanTable ← GeneratePlan(q,κ)
6         ExecutePlan(PlanTable) // executes the enrichment
              plan and updates the state tables.
7         κ ← κ+1
8         RefreshAnswer(IMV)
9     Return IMV
```

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, TAGDB includes the following three (special purpose) UDFs to support the query semantics of §3.2.

Determinization UDF (*DET*()): takes the state of the tuple as input and outputs determinized representation of the tuple, *i.e.*, a tag, for a derived attribute. By default, in TAGDB, *DET*() UDF performs threshold-based determinization, but it can use other strategies (*e.g.*, Top-1 [97] and quality maximization [64, 70]), as explained in §3.1. *DET*() UDF, internally, calls a ***combiner function*** that combines the output of multiple enrichment functions of a derived attribute, see §2.

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 (see §3.2). For example, consider tuple t_1 of `TweetData` table and the query of Figure 1. Suppose the determinized representation of the tuple in `topic` column 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.

4.5.1 ***Query Setup.*** Query setup performs the following tasks: (*i*) query rewrite to deal with determinized representation and (*ii*)

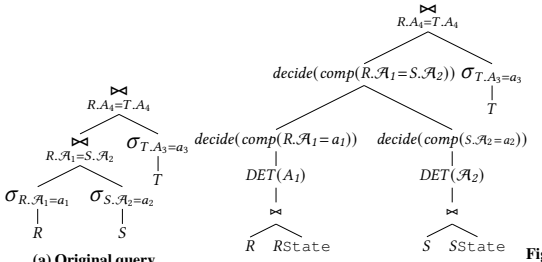
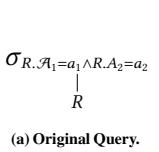


Figure 5: Query rewriting process.



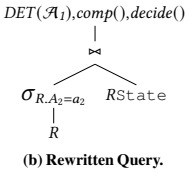
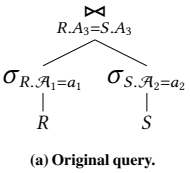


Figure 6: Query rewriting processing, showing the predicate on the fixed attribute A_2 is pushdown.



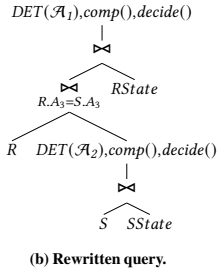


Figure 7: Join queries with join condition on fixed attribute A_3.

computation of the initial set of results, without further enriching the data. We explain the query rewrite process using a sample query shown in Figure 5a that involves three relations $R(\mathcal{A}_1, A_4)$, $S(\mathcal{A}_2)$, $T(A_3, A_4)$. The corresponding rewritten query is shown in Figure 5b. In general, query rewriting process contains the following stages:

(1) In order to find relations that require determinization, TAGDB lists all relations that have query predicates/projection on derived attributes (*e.g.*, R and S in our example Figure 5b).

(2) To find the determinized representation of tuples, the relations are first joined with their corresponding `State` tables (*e.g.*, R is joined with `RState` and S is joined with `SState`) and then passed to *DET*() UDF (*e.g.*, $DET(R.\mathcal{A}_1)$ and $DET(S.\mathcal{A}_2)$).

(3) Since determinized representations of tuples cannot be matched directly against query predicates, tuples are passed through *comp*() UDF that evaluates the query predicate using predicate evaluation logic, as given in §3.2 (*e.g.*, $comp(R.\mathcal{A}_1 = a_1)$ and $comp(S.\mathcal{A}_2 = a_2)$).

(4) *decide*() UDF is placed as a filter on top of the output of *comp*() UDF, to decide which tuples should be processed by the next operator (*e.g.*, $decide(comp(R.\mathcal{A}_1 = a_1))$, $decide(comp(S.\mathcal{A}_2 = a_2))$, and $decide(comp(R.\mathcal{A}_1 = S.\mathcal{A}_2))$.

(5) For a join involving derived attributes, the join condition is replaced by a filter using *comp*() and *decide*() UDFs (*e.g.*, $decide(comp(R.\mathcal{A}_1 = S.\mathcal{A}_2))$.

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 thus, do not need rewriting using *comp*().

Example 4.1. We show query rewrite using the above steps by TAGDB for the query shown in Figure 1. This query is interested in tuples (of `TweetData`) that have conditions on derived attributes, as `topic` = social media and `sentiment` = positive. The following command presents the rewritten query by TAGDB that is *directly* executed on PostgreSQL. PostgreSQL can select its underlying optimization criteria that, however, does not affect the query semantics of TAGDB.

```
CREATE INCREMENTAL MATERIALIZED VIEW mat_view AS
 SELECT TweetObject, location, TweetTime, topic, sentiment
 FROM (SELECT * FROM TweetData
  WHERE TweetTime BETWEEN ('2020-02-20', '2020-02-21') as R
  NATURAL JOIN TweetDataState AS RState
 WHERE Decide
      (Comp(Det(SentimentStateOutput), 'sentiment=positive')) AND
 Decide(Comp(Det(TopicStateOutput), 'topic=social media'))
```

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

- **Predicates on fixed attributes.** TAGDB pushes down the predicates on the fixed attributes in the query tree. Consider the query shown in Figure 6a, where the selection conditions on a relation involve both fixed and derived attributes (*i.e.*, $R.\mathcal{A}_1 = a_1 \wedge R.A_2 = a_2$). It can be seen in the rewritten query (Figure 6b) that the predicate on fixed attributes, *i.e.*, $R.A_2 = a_2$, is pushed down and is placed below the join of R and `RState`. Here, filtering of tuples of R based on the predicates on fixed attributes before the join with state table ensures that the join result (with `RState`) becomes small; hence, undesired tuples are eliminated before enrichment.
- **Join condition on fixed attributes.** If a query involves a join over only fixed attributes of tables, then TAGDB performs such a join before performing the determinization of the tables involved in the join (*i.e.*, prior to join of R and `RState` tables). Consider the query in Figure 7a, where the join condition is on a fixed attribute. In the rewritten query (see Figure 7b), TAGDB joins R with S before joining it with `RState` and filters out the tuples of R that do not match the condition $R.A_3 = S.A_3$. This optimization is useful, when fewer tuples of R joins with tuples of S (*i.e.*, low join cardinality), as it reduces the number of tuples in R to be enriched.

Incremental Materialized View (IMV). Since TAGDB 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, TAGDB creates an incremental materialized view (IMV) [4, 20, 60, 69] on the rewritten query (as shown in Example 5.3.1). 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 evaluating 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 much more efficient. In the first epoch, the initial query results — produced by the rewritten query depending on the current state of tuples without any further enrichment — are stored in IMV, and in later epochs, delta answers are appended/updated/deleted to/from IMV.

4.5.2 *Enrichment Plan Generation.*

An *enrichment plan* for a query in an epoch (say e_κ) consists of a set of ⟨tuple, enrichment function⟩ pairs, where the enrichment function will be executed on the tuple in the epoch e_κ. 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 TAGDB or provided by user as discussed in §4.3), while the ***sampling-based*** (**SB**) method *randomly* selects pairs of tuples and enrichment functions.

RelName	TID	AttrName	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 11: `PlanSpaceTable`. *`benefit` column only used in benefit-based method.

RelName	TID	AttrName	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 12: `PlanTable`.

The effectiveness of a plan generation strategy is measured using the progressive score defined in Equation §2 (in §5 we experimentally compare plan generation methods based on the progressive score). Importantly, in TAGDB, users can incorporate their own plan generation methods by implementing functions *GetNextFunction*() and *SelectBestFeasiblePlan*(), shown in Algorithm 2.

TAGDB maintains a `PlanSpaceTable` (see Table 11 as an example) to guide the plan generation for a query q. Rows in `PlanSpaceTable` correspond to all relation names ($Name(R_i)$) included in q, their tuple identifiers ($ID(t_\ell)$), and derived attributes' name ($Name(\mathcal{A}_j)$) with $\langle f_m, benefit\rangle$, where f_m is the next function that should be executed on the tuple t_ℓ and its *benefit* (used only in BB method). Note that the number of rows in `PlanSpaceTable` can be large, later we discuss how to limit that. `PlanSpaceTable` represents a subset of all possible enrichment plans for the query q. A subset of entries in `PlanSpaceTable` is selected for execution during a given epoch, and we refer to this subset as `PlanTable` for the epoch. The cost of the selected plan is the summation of the cost of enrichment functions mentioned in rows of `PlanTable`. Note that for the plan to be valid (*i.e.*, executable during the epoch), the cost of the selected plan *must* be smaller than the epoch duration (δ). The selection of rows in `PlanTable` from `PlanSpaceTable` is constrained to ensure the validity of the selected plan.

Algorithm 2 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 the BB method, our goal is to pick a subset of tuples, thereby maximizing the total benefit and bounding the total cost by δ. This problem can be seen as a bounded weight resource maximization knapsack problem [23], which is NP-hard. Thus, we use Weighted Shortest Processing Time first (WSPT) heuristic-based approach [23] to select a set of tuples from the `PlanSpaceTable`. Using WSPT, TAGDB, 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 the `PlanSpaceTable`. While any sampling can be used (*e.g.*, uniform random, stratified random, or variational sub-sampling as used in [9, 76]), TAGDB uses uniform random sampling.

Algorithm 2: TAGDB plan generation.

Inputs: $\langle q, \delta, \texttt{state}(R_i), type, \text{DT}, \kappa\rangle$. // q: a query on relations $R_1,...,R_n$. δ: epoch duration, `state`(R_i): the state table of R_i. *type*: the type of plan generation — sampling or benefit, DT: `DecisionTable` used only in benefit-based plan generation, and κ: epoch id.

Outputs: Enrichment plan, *i.e.*, `PlanTable`, for q for epoch κ.

Modules: (*i*) ***GetDerivedAttributeQuery***$(*)$: retrieves derived attributes of relations involved in a query. (*ii*) ***GetNextFunction***$(*,*)$: retrieves the next best function to execute (*iii*) ***Name***$(*)$: retuns the name of relation or attribute. (*iv*) ***ID***$(*)$: returns the identity. (*v*) ***GetTopTuples***$(*,*)$: returns a set of tuples with highest benefit values (*vi*) ***Sort***$(*,*)$: returns a sorted relation (*vii*) ***GetRANDSample***$(*,*)$: returns a random sample

Variables: f_m: enrichment function. ***benefit***: benefit of enrichment.

```
1  Function GeneratePlan(κ) begin
2      if κ = 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 ∀R_i ∈ q do
8          attribute_list[] ← GetDerivedAttributeQuery(q(R_i))
9          for ∀(j,ℓ) ∈ [A_j ∈ attribute_list[]] × [t_ℓ ∈ R_i] do
10             if type = benefit then
11                 f_m, benefit ← GetNextFunction(state(t_ℓ.A_j), DT)
12                 Append ⟨Name(R_i), ID(t_ℓ), Name(A_j), f_m, benefit⟩ to
                     PlanSpaceTable
               else
13                 f_m ← GetNextFunction(state(t_ℓ.A_j))
14                 Append ⟨Name(R_i), ID(t_ℓ), Name(A_j), f_m⟩ 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 ∀⟨Name(R_i), ID(t_ℓ), Name(A_j), f_m, benefit⟩ ∈ PlanTable do
23         if type = benefit then
               f_p, benefit ← GetNextFunction(state(t_ℓ.A_j), DT)
24         else f_p ← GetNextFunction(state(t_ℓ.A_j))
25         if f_p = ∅ then Delete ⟨Name(R_i), ID(t_ℓ), Name(A_j), f_m, benefit⟩
             from PlanSpaceTable
26         else Update the function for Name(R_i), ID(t_ℓ), Name(A_j) to f_p in
             PlanSpaceTable.
27     Return PlanSpaceTable
```

Plan Space Update (Lines 21-27). Since the execution of a plan in an epoch alters the state of the data, TAGDB needs to recompute `PlanSpaceTable` in the next epoch. However, `PlanSpaceTable` must not be computed completely from scratch in each epoch, since it incurs overhead. Thus, TAGDB updates only those tuples of `PlanSpaceTable` that were part of `PlanTable` in the previous epoch, as for all tuples in `PlanTable`, a new next function f_p and the corresponding benefit value is computed (Line 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 §4.5.1). 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`. *E.g.*, in Figure 7b, tuples of R added to `PlanSpaceTable` for possible enrichment are only those tuples that satisfy the join condition on fixed attribute A_3 ($R.A_3 = S.A_3$). Similarly, in Figure 6b, tuples of R that satisfy filtering condition $R.A_2 = a_2$ are considered for enrichment.

4.5.3 *Enrichment Plan Execution and State Update.*

Enrichment plans obtained from Algorithm 2 (mentioned in §4.5.2) are executed using a UDF, called ***driver UDF***. The driver UDF can execute any enrichment function supported by TAGDB. 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, TAGDB 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[15] enriches tuples of `TweetData` table based on `PlanTable`.

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

4.5.4 *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 (§4.5.1), the update of state tables triggers an (automatic) update of IMV. *E.g.*, if the state of x tuples changes from the previous epoch in `TweetDataState` table, it might result in a change to determinized representations of these x tuples. Thus, some of these tuples that were previously filtered, may be passed by *comp*() UDF in the query tree. Similarly, some tuples that were earlier passed by *comp*() UDF, may be filtered out due to the new determinized representation. Note that IMV supports incremental updates of any monotonic SPJ and aggregation query.

Users can fetch complete answers at the end of an epoch by querying the IMV. If the complete answer set is large, TAGDB allows users to retrieve delta changes of the answers, *i.e.*, inserted/deleted/updated tuples from the previous epoch. TAGDB provides the delta answers to users through the delta tables[16] that are maintained by IMV. However, 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 of TAGDB), since the query processing in TAGDB is not demand-driven, as in traditional SQL databases.

4.6 Parallel Query Execution

The algorithm of query execution as described in §4.5 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 1.

[15] We rewrite the query while passing it to PostgreSQL, by expanding '*' with all the column names of `TweetData` and `FunctionMetadata`.

[16] Delta tables store the delta results that are computed from a delta relational algebra expression [3, 69] and they are used to update the materialized view incrementally.

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 1 is being executed on a machine with 4 CPU cores. TAGDB 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 (
    '2020-02-20 00:00:00', '2020-02-20 06:00:00');
sq2: SELECT ... FROM TweetData
     WHERE sentiment = 'positive' AND topic
     = 'social media' AND tweet_time BETWEEN (
    '2020-02-20 06:00:00', '2020-02-20 12:00:00');
sq3: SELECT ... FROM TweetData
     WHERE sentiment = 'positive' AND topic
     = 'social media' AND tweet_time BETWEEN (
    '2020-02-20 12:00:00', '2020-02-20 18:00:00');
sq4: SELECT ... FROM TweetData
     WHERE sentiment = 'positive' AND topic
     = 'social media' AND tweet_time BETWEEN (
    '2020-02-20 18:00:00', '2020-02-21 00:00:00');
```

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 *sq*1 and *sq*2 finished their data enrichment phase in 5 seconds and 6 seconds respectively.

Relation	#tuples	Size(GB)	Derived attributes(domain size)	Functions used
TweetData	11M	10.5	sentiment(3)	GNB, KNN, SVM, MLP
			topic(40)	GNB, KNN, LDA, LR
ImageNet [38]	100K	19	ObjectClass(100)	DT, GNB, RF, MLP
MultiPie [87]	100K	16.9	gender(2), expression(5)	DT, GNB, KNN, MLP
Synthetic Dataset	100M	35	$\mathcal{A}_1$(5)	Functions: $f_1, \ldots, f_{10}$.

Table 13: Datasets used in experiments.

ID	Query
Q1	SELECT * from **ImageNet** where ObjectClass = 2 where ImageID between (20000,30000)
Q2	SELECT * from **MultiPie** where gender = 1 and CameraID between (5,12)
Q3	SELECT tid, UserID, TweetObject, location, TweetTime from **TweetData** where sentiment = 1 and TweetTime between('2020-02-20','2020-02-21')
Q4	SELECT * from **SyntheticData** where $\mathcal{A}_1$ = 1 and A_2 between (1, 100000)
Q5	SELECT * from **MultiPie** where gender = 1 and expression =2 and CameraID between (5,12)
Q6	SELECT * from **TweetData** T1, **TweetData** T2 where T1.sentiment = T2.sentiment and T1.TweetTime between('2020-02-20','2020-02-21') and T2.TweetTime between ('2020-02-21','2020-02-22')
Q7	SELECT * from **TweetData** T1, State S where T1.location = S.city and S.state='California' and T1.sentiment = 1 and T1.TweetTime between('2020-02-20','2020-02-21')
Q8	SELECT gender, count(*) from **MultiPie** where CameraID between (5,12) group by gender
Q9	SELECT ObjectClass, count(*) from **ImageNet** where ImageID between (20000,30000) group by ObjectClass

Table 14: Queries used.

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 [37]. 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 TAGDB 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. TAGDB 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 1, 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. TAGDB 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. TAGDB re-splits the queries as discussed earlier in straggler mitigation and start them in the new epoch.

5 EXPERIMENTAL EVALUATION

We evaluate TAGDB from the following perspectives: (*i*) progressive quality improvement achieved during query execution and (*ii*) performance overhead of TAGDB 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), and the parameters used for determinization. For performance, we study the impact of various optimizations.

5.1 Experimental Setup

Datasets. For the experiments, we have used four datasets to evaluate the ***performance of* TagDB**. Table 13 shows the size of the datasets and the derived attributes and enrichment functions used in them. The datasets corresponded to: (*i*) TweetData collected using Twitter APIs, (*ii*) ImageNet [38] dataset consisting of images of objects, (*iii*) MultiPie [87] dataset consisting of facial images, and (*iv*) a large synthetic dataset for evaluating the scalability of TAGDB.[17]

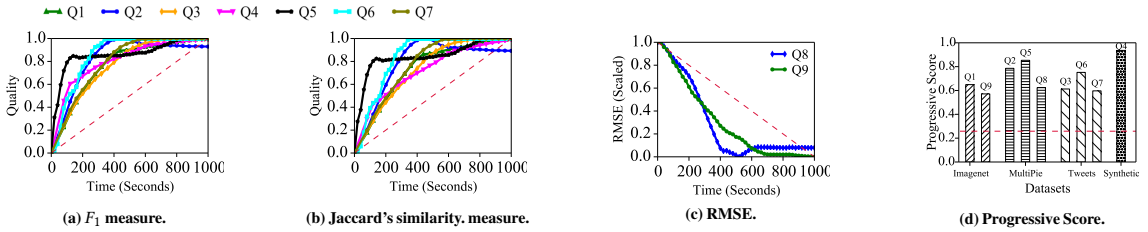
Enrichment Functions. In the first three datasets, the following probabilistic classifiers were used as enrichment functions: Gaussian Naïve Bayes (GNB), Decision Tree (DT), Support Vector Machine (SVM), k-Nearest Neighbor (KNN), Multi-Layered perceptron (MLP), Linear Discriminant Analysis (LDA), Logistic Regression (LR), and Random Forest (RF); see Table 13. The GNB classifier was calibrated using isotonic-regression model [99] and all the other classifiers were calibrated using Platt's sigmoid model [79] during cross-validation. After calibration, each classifier output can be considered as a probability distribution [79]. For the synthetic dataset, we used synthetic enrichment functions where the functions are each associated with varying levels of quality and cost as will be discussed in the experiments. When the quality of a synthetic function is 0.8, the function predicts the correct value of a tuple 80% of the time.

Unless specified explicitly, we used a fixed ***threshold-based determinization strategy*** (see §3.1) for each derived attribute in the experiments, where the threshold is fixed to be the inverse of domain size (Exp 4 shows precision/recall variation of answer for different threshold values). *E.g.*, for an attribute with three possible values, the threshold is set to be 0.33. Further `weighted average` is used as a combiner function, where weights are proportional to the quality of enrichment functions learned based on the mechanism of §2.

Queries. As shown in Table 14, we selected nine queries, where ***Q1-Q4*** are ***selection queries*** one for each of the four datasets, ***Q5*** is a ***conjunctive query*** having two derived attributes on MultiPie dataset, ***Q6-Q7*** are ***join queries*** on TweetData (where Q6 involves join condition on a derived attribute while Q7's join condition is on a fixed attribute), and ***Q8-Q9*** are ***aggregation queries*** on MultiPie and ImageNet datasets, involving the group by condition on a derived attribute. The number of possible groups in Q8 is low (*i.e.*, 2), while in Q9, it is high (*i.e.*, 100). The ***epoch size*** for all queries in all experiments (except Exp 5) was ***20 seconds***.

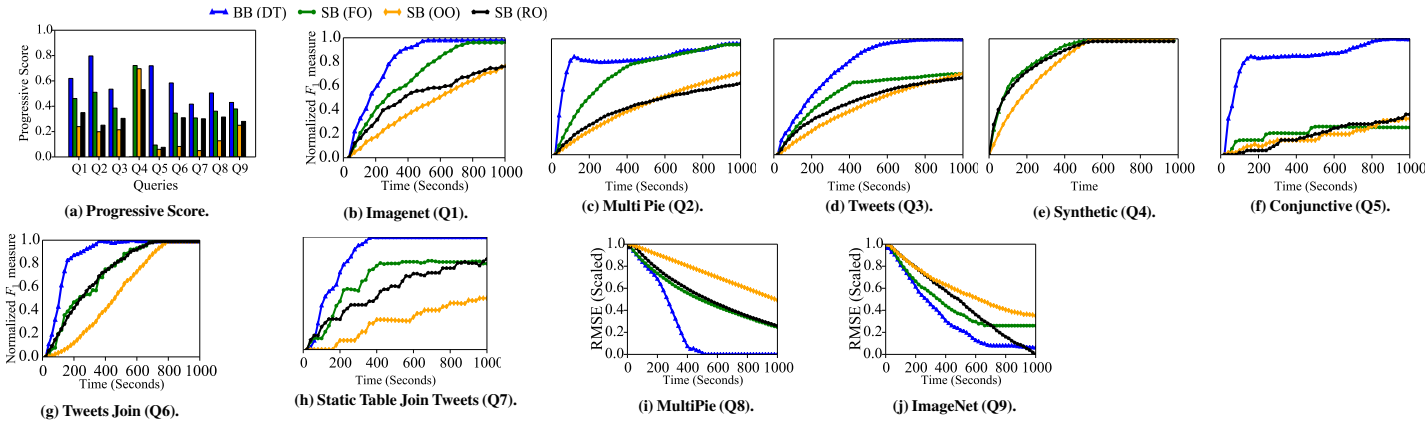
Plan Generation Strategies. For experiments, the following four different plan generation strategies were used: (*i*) *Benefit-based approach using Decision Table* (**BB(DT)**): that selects a set of ⟨tuple, function⟩ pairs with the highest benefit value based on the decision table. (*ii*) *Sample-based 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, where *quality* is the accuracy of prediction and *cost* is

[17] Though TAGDB is deployed to build IoT applications in our campus, we restricted our experiments to publicly available data sets and functions shared by us on Github [1]. The campus data set contains private location information that is not available without IRB (Institutional Review Board) approvals, and hence cannot be shared with the larger community.



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

Figure 8: Exp 1. 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 9: Exp 2. Different plan generation strategies.

the average execution cost per tuple, 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.

5.2 Experimental Results

The experiments were performed on an AWS server with 16core, Intel Xeon CPU with 2.50GHz processor, 64GB RAM, and 1TB SSD. **Exp 1-4** are progressiveness experiments, **Exp 5-6** are performance experiments and **Exp** 7 is an experiment for query parameter selection.

Exp 1: Progressiveness of different queries. Figure 8 evaluates TAGDB 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 8a 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 Figure8b. For aggregation queries Q8 and Q9, the quality is measured using ***normalized root mean square error*** (RMSE) (Figure 8c). We plot normalized measures as a function of time to emphasize the rate at which TAGDB 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 8a and 8c, we observe that TAGDB achieves a very high quality improvement (95% of maximum F_1 measure and 95% reduction of RMSE) within the first few epochs of query execution. Also, TAGDB 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 8). Since for queries with blocking operators (joins and group-by aggregation) all tuples need to be enriched completely before evaluating the join or aggregation operator, such strategy cannot be devised.

Figure 8d shows the progressive score (as defined in Equation 2) achieved for each query. Observe that TAGDB achieves a high progressive score for all queries as compared to the possible iterative strategy (dotted line in Figure 8d). Table 15 shows the query execution time for Q1-Q9, when they are executed after complete enrichment of tuples. Comparing Table 15 with Figure 8, we observe that TAGDB returns high quality results within a few epochs without requiring users to wait for complete enrichment.

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

Table 15: Exp 1. Query time without progressive execution.

Exp 2: Effect of Different Plan Generation Strategies. Figure 9 studies different plan generation strategies and their impact on progressiveness. Figure 9a plots progressive score of nine queries Q1-Q9 on ImageNet, Tweets and MultiPie datasets.

Figures 9a - 9j, show BB(DT) performs better than sampling based approaches highlighting that the decision table learned by TAGDB 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 will

use BB(DT) as a plan generation strategy, except for synthetic dataset, for which we do not store benefit values.

Exp 3: Impact of enrichment functions. 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 TAGDB. 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 10): (*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 11). 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 11a), 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 10b. From Figures 9b, 9c, and 9d 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 the highest positive curvature in quality-cost graph as compared to the ImageNet and TweetData functions.

Exp 4: Impact of Threshold in Determinization Function. In the previous experiments, we used a threshold-based determinization function, where a threshold is set up for each derived attribute of a relation (as described in §5.1). This experiment evaluates how the quality of answers is affected when thresholds are varied in such a determinization function. Figure 12 shows the quality of the query answer achieved after executing Q1 and Q3 for fifty epochs using different thresholds. We observe that the precision of the query answer increases with increase in threshold as higher threshold reduces false positive tuples from the answer set. The recall on the other hand decreases with increase in threshold as many of the true positive tuples get eliminated from the answer set.

Exp 5: System Overhead. This experiment measures the following two overheads incurred by TAGDB:

(***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. Note that majority of the time for benefit calculation (*i.e.*, 50s) was spent in the first epoch due to the calculation of benefit of all tuples satisfying query predicates on fixed attributes.

(***ii***) ***Storage overhead*****:** measures the size of all temporary tables and IMV, used during query processing to compare against the size of TAGDB tables. Table 16 shows the maximum storage overheads of the benefit table, plan table, and IMV at any epoch. Observe that this overhead is much lower than the size of TAGDB tables (given in Table 13). Table 17 shows the size of state tables that is also significantly lower than TAGDB tables. In Table 17, the third column shows the improvement in storage space due to state cutoff (§4.4). Since the domain size of `ObjectClass` in the ImageNet dataset is 100, the state cutoff representation reduced the size of state table from 6.8 GB (stored using default representation) to 0.94 GB (stored using state cutoff). Also, the combined size of all metadata tables, (*i.e.*, `RelationMetadata`, `FunctionMetadata`, `FunctionFamily`, and `DecisionTable`) is less than 10 MB.

Exp 6: Impact of Optimizations. §4.5.1 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 investigate these optimizations. Table 18 presents the size of benefit table (*i.e.*, the tuples considered for enrichment) with and without these optimizations. From Table 18, it is clear that the benefit table size reduces significantly due to optimizations, (and hence the complexity of the enrichment plan generation reduces).

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 16: Exp 5. 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 17: Exp 5. Storage cost of state.

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

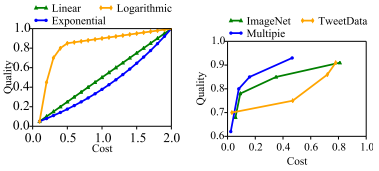
Table 18: Exp 6. Impact of optimization.

Exp 7: Effect of Epoch Size. This experiment shows the impact of different epoch sizes (in seconds) on the quality of answers and the overhead of TAGDB, on a randomly selected query Q2. Figure 14a shows time to reach (TTR) 90% quality for Q2 with respect to epoch size. Observe that as the epoch size reduces from 50 to 20, TTR also reduces, since smaller epoch size causes more frequent plan generation and results in faster improvement of answer quality. However, reducing epoch size much more (*i.e.*, 5 or 10), TTR starts to increase, since the time overhead of frequent plan generation overshadowed the improvement in quality achieved (Figure 14a). Figure 14b shows that the time overhead decreases with the increase in epoch size, since for larger epochs, TAGDB has to perform less frequent plan generation as compared to smaller epochs.

6 RELATED WORK

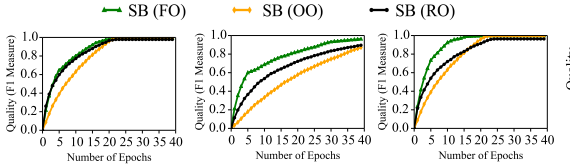
Progressive and query-driven approaches have been studied in several domains: entity resolution [11, 13, 14, 73, 94], crowd-sourced data processing tasks [15, 74, 75], online schema matching [67, 81], and probabilistic databases [35, 83, 86]. Incremental query processing and view maintenance were studied in [4, 20, 60, 69, 90]. However, none of them considered progressive query processing along with enrichment.

Systems for supporting ML using databases (*e.g.*, Apache MADlib [46], Bismarck [41], RIOT [102], SystemML [21, 43], SimSQL [24])



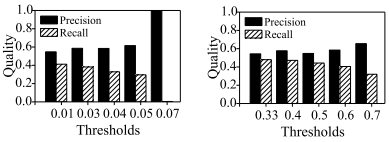
(a) Synthetic Dataset. (b) Real Dataset Functions.

Figure 10: Exp 3. Function correlations.



(a) Linear correlation. (b) Logarithmic correlation. (c) Exponential correlation.

Figure 11: Exp 3. Comparison of plan generation strategies in synthetic dataset.



(a) Imagenet (Q1). (b) Tweets (Q3).

Figure 12: Exp 4. Effect of different thresholds.

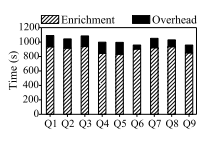
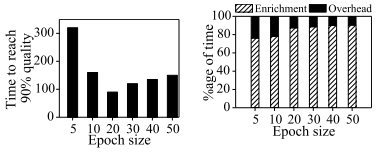


Figure 13: Exp 5. Time overhead.



(a) TTR 90% (Q2). (b) Overhead (Q2).

Figure 14: Exp 7. Effect of different epoch sizes.

are designed to learn ML models inside or on top of database systems; TAGDB is related to them, but complementary and focuses on optimally enriching data at query time, using models learnt by these systems. Also, recent systems address the problem of model selection and optimal plan generation for ML inference queries [33, 57]; however, they do not support progressive query execution. **Expensive predicate optimization** problem optimizes queries containing expensive predicates [17, 28, 47, 54, 56, 61, 65, 66]. [54, 61, 65] address *optimization of multi-version predicates*, by using less expensive predicates to filter objects before evaluating expensive predicates. However, [54, 61, 65] considered predicates to be static and deterministic. In contrast, TAGDB supports both deterministic and non-deterministic enrichment functions/predicates.

7 CONCLUSION & FUTURE DIRECTIONS

We introduce the architecture and implementation of TAGDB, a data management system that supports policy-based enrichment all through the data processing pipeline, at ingestion, intermittently based on events, or during query execution. TAGDB 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. TAGDB 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 TAGDB.

Current TAGDB implementation uses determinized representation of attributes that require enrichment. An alternate would be to use *possible-world* semantics [10, 35, 50, 53, 72, 83, 86]. Though query processing in probabilistic databases have been extensively studied using several techniques such as Monte Carlo algorithms [16, 53, 98], top-k processing [83], sequential databases [55, 63, 84], specialized indexes [30, 59, 88] and materialized views [36, 85], adopting such techniques in TAGDB would require modifying them to dynamic situations, where existential probabilities of tuples may change as a result of enrichment and queries are evaluated progressively.

REFERENCES

[1] TagDB code. https://github.com/TagDB20/tagdb.
[2] Apache kafka. https://kafka.apache.org/23/documentation/streams/.
[3] Incremental view maintenance development for postgresql. https://github.com/sraoss/pgsql-ivm.
[4] Incremental view maintenance for postgresql. https://wiki.postgresql.org/wiki/Incremental_View_Maintenance.
[5] Machine learning extensions library for python. http://rasbt.github.io/mlxtend/.
[6] Tensorflow. https://www.tensorflow.org/.
[7] Timescale. https://www.timescale.com/.
[8] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. *SIGMOD Rec.*, 28(2):574–576, June 1999.
[9] 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.
[10] 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.
[11] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 7(11):999–1010, 2014.
[12] Y. Altowim and S. Mehrotra. Parallel progressive approach to entity resolution using mapreduce. In *ICDE*, pages 909–920. IEEE Computer Society, 2017.
[13] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. *PVLDB*, 6(14):1846–1857, 2013.
[14] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3):120–131, 2015.
[15] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD Conference*, pages 241–252. ACM, 2013.
[16] S. Arumugam, R. Jampani, L. L. Perez, F. Xu, C. M. Jermaine, and P. J. Haas. MCDB-R: risk analysis in the database. *Proc. VLDB Endow.*, 3(1):782–793, 2010.
[17] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000.
[18] 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.
[19] L. Berrada, A. Zisserman, and M. P. Kumar. Smooth loss functions for deep top-k classification. *arXiv preprint arXiv:1802.07595*, 2018.
[20] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.
[21] 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.
[22] A. P. Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 1997.
[23] P. Brucker. *Scheduling Algorithms*. Springer Publishing Company, 2010.
[24] 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.
[25] 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.
[26] S. Chaudhuri, G. Das, and V. R. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2):9, 2007.
[27] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD Conference*, pages 263–274. ACM Press, 1999.
[28] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 1999.
[29] J. Chen, C. Xiong, and J. Callan. An empirical study of learning to rank for entity search. In *SIGIR*, pages 737–740. ACM, 2016.
[30] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, pages 876–887. Morgan Kaufmann, 2004.
[31] W. Cheng, E. Hüllermeier, and K. J. Dembczynski. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, pages 279–286, 2010.
[32] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, pages 313–328. USENIX Association, 2010.
[33] 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.
[34] 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.
[35] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, Oct. 2007.
[36] N. N. Dalvi, C. Ré, and D. Suciu. Queries and materialized views on probabilistic databases. *J. Comput. Syst. Sci.*, 77(3):473–490, 2011.

[37] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[38] 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.

[39] 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.

[40] K. Eykholt, A. Prakash, and B. Mozafari. Ensuring authorized updates in multi-user database-backed applications. In *USENIX Security Symposium*, pages 1445–1462. USENIX Association, 2017.

[41] 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.

[42] J. H. Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[43] 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.

[44] R. Hai, S. Geisler, and C. Quix. Constance: An intelligent data lake system. In *SIGMOD Conference*, pages 2097–2100. ACM, 2016.

[45] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.

[46] 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.

[47] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 1993.

[48] 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.

[49] 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.

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

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

[52] 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.

[53] 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.

[54] 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.

[55] B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *SIGMOD Conference*, pages 455–468. ACM, 2009.

[56] 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.

[57] 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.

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

[59] H. Kimura, S. Madden, and S. B. Zdonik. Upi: A primary index for uncertain databases. *Proceedings of the VLDB Endowment*, 3(1-2):630–637, 2010.

[60] 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.

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

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

[63] J. Letchner, C. Ré, M. Balazinska, and M. Philipose. Access methods for markovian streams. In *ICDE*, pages 246–257. IEEE Computer Society, 2009.

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

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

[66] 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.

[67] 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.

[68] 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.

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

[70] 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.

[71] 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.

[72] L. J. Orr, M. Balazinska, and D. Suciu. Entropydb: a probabilistic approach to approximate query processing. *VLDB J.*, 29(1):539–567, 2020.

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

[74] 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.

[75] 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.

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

[77] Y. Park, A. S. Tajik, M. J. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *SIGMOD Conference*, pages 587–602. ACM, 2017.

[78] 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.

[79] 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.

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

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

[82] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD Conference*, pages 275–286. ACM, 2002.

[83] 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.

[84] C. Ré, J. Letchner, M. Balazinska, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD Conference*, pages 715–728. ACM, 2008.

[85] A. D. Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, pages 1023–1032. IEEE Computer Society, 2008.

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

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

[88] S. Singh, C. Mayfield, S. Prabhakar, R. Shah, and S. E. Hambrusch. Indexing uncertain categorical data. In *ICDE*, pages 616–625. IEEE Computer Society, 2007.

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

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

[91] I. G. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino. Data wrangling: The challenging yourney from the wild to the lake. In *CIDR*. www.cidrdb.org, 2015.

[92] 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.

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

[94] 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.

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

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

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

[98] J. Xu, Z. Zhang, A. K. H. Tung, and G. Yu. Efficient and effective similarity search over probabilistic data based on earth mover's distance. *VLDB J.*, 21(4):535–559, 2012.

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

[100] 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.

[101] 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.

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

8 APPENDIX

8.1 Case Study

This section provides a case study to motivate the need of our proposed TAGDB system, that *progressively enriches* data at query time.

Smart Campus Case Study. We consider a motivating application in which a university campus supports a variety of smart data services such as real-time queue size detection in different food courts, occupancy analysis to study the usage level of different public places such as libraries, class rooms, cafeterias, study rooms etc. Such analysis are used to determine the room occupancy as a function of time and events, determine how space organization impacts interactions amongst occupants etc. The case study is based on our experience in building a smart campus with variety of applications ranging from real-time services to offline analysis over the past four years.

This campus consists of 162 buildings wherein a diverse set of sensors, such as Wi-Fi access points, Bluetooth beacons, surveillance cameras, HVAC sensors, and proximity sensors are installed to support a variety of data services. The data generation rate from these sensors are as follows: ~30,000 tuples of WiFi connectivity data per second from the WiFi access points, ~100,000 tuples of Bluetooth beacon connectivity data per second, and ~4000 images per second (with a size of 360 MB) from the cameras. The data enrichment tasks from the end applications are as follows: performing building level localization from the WiFi connectivity data, performing floor level localization from the beacon data and performing most accurate room level localization from the camera data.

Following enrichment tasks are required to be performed on the collected data to derive accurate localization of the users. For each WiFi Access Points in the campus, we store the coverage of the access points in a *WiFi_Coverage* table. This table stores the coverage of each access points in terms of the buildings/regions of a building in which the access point is accessible, *e.g.*, ⟨⟨WiFi-AP1, Region-1 ⟩, ⟨WiFi-AP1, Region-2 ⟩⟩. Enrichment of a tuple of WiFi access point connectivity data involves looking up this coverage table to find out the list of locations (*i.e.*, Region-1, Region-2) that are covered by the access point (*i.e.*,WiFi-AP1) and generate a presence tuple corresponding to each of the locations.

The beacon connectivity tuples are stored as follows: ⟨ UUID, MajorID, MinorID, DeviceID, RSSI, timestamp⟩ where the attributes UUID, MajorID and MinorID are used to identify a beacon uniquely. UUID attribute stores the Universally Unique IDentifier of the beacon. MajorID and MinorID are additional identifiers that are used to differentiate the locations of the beacons (*i.e.*, all the beacons in the same floor are assigned the same major ID, and the floor is further subdivided into four regions where the beacons of each regions are assigned a minor ID. DeviceID is the unique identifier of the device that is connected to the beacon (*i.e.*, Bluetooth MAC-address of a mobile device), RSSI represents received signal strength indicator and timestamp stores the time at which the data was captured.

Enriching a single tuple of WiFi connectivity data to generate the building level localization of a user takes ~10 milliseconds to complete, whereas generating the floor level localization from a tuple of beacon connectivity data takes ~40 milliseconds to complete (by extracting the signal strength and determining the most probable floor location of the user based on the signal to distance ratio of the beacon). The enrichment of an image to determine the most accurate room level location of a user takes ~4 seconds to complete (due to the feature extraction of the frame and the execution of a Neural-Network classifier on the feature-vector to determine the person inside the image).

If a system decides to perform all the enrichment at the ingestion time, then it will take the following amount of time: $30{,}000 \cdot 10$ ms + $100{,}000 \cdot 40$ ms + $4000 \cdot 4$ s = 20,300 s ≃ 5.63 hours to analyze the data generated in only one second. Even with the horizontal scaling (by adding more machines to process the incoming data in a parallel manner) and/or vertical scaling (by adding sophisticated GPGPUs to perform machine learning tasks faster), such numbers can not be reduced considerably.

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 19: Model table for a multi-layered perceptron classifier.

From the above case study, it is clearly visible that data enrichment can not be performed at the data ingestion time. Alternatively, if a system chooses to periodically enrich the data similar to batch updates in the *Online-Analytical Processing* (OLAP) systems (*i.e.*, once in every 12 hours) it is impossible to complete enrichment on such a large amount of data. Hence, we need a system that can perform enrichment at the query time only at the context of a query. This ensures that ingestion latency remains low as none of the enrichment is performed at the data ingestion time and enrichment is only performed on the useful part of the data that is being used in the queries.

Furthermore, even a query time approach will not scale if we take the traditional approach of query processing where we perform the data enrichment completely in the context of the query and then execute the query on the enriched data to return the results. In this scenario, the user has to wait for several hours (≃ 5.63 hours as shown in the above case study) in order to receive the answer of a query which is unacceptable. Hence, we need an iterative approach of query answering in which the system returns a quick and low quality answer to the user (very much in the realm of Approximate Query Processing style query answering) and keeps improving the quality of the answer progressively by performing more and more enrichment on the data with respect to time. The benefit of such a progressive approach of query answering is many fold: it reduces the query response time significantly, since result of a query gets improved with respect to time, users can choose to stop the query processing any time they want as soon as a satisfactory level of quality is achieved by the answer.

8.2 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 19. 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[46].