Supporting Complex Query Time Enrichment For Analytics

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

ABSTRACT

Several application domains require data to be enriched prior to its use. Data enrichment is often performed using expensive machine learning models to interpret low-level data (*e.g.*, models for face detection) into semantically meaningful observation. Collecting and enriching data offline before loading it to a database is infeasible if one desires online analysis on data as it arrives. Enriching data on the fly at insertion could result in redundant work (if applications require only a fraction of data enriched) and could result in a bottleneck (if enrichment functions are expensive). Any scalable solution requires enrichment during query processing. This paper explores two different architectures for integrating enrichment into query processing – a loosely coupled approach wherein enrichment is performed outside of the DBMS and a tightly coupled approach wherein it is performed within the DBMS. The paper addresses the challenges of increased query latency due to query time enrichment.

1 INTRODUCTION

Organizations, today, have access to potentially limitless data sources in the form of web data repositories, social media posts, and continuously generated sensory data [7]. Such data is often low-level and needs to be enriched to be useful for analysis. Functions used to *transform* or *enrich* data (called *enrichment functions* in this paper) could consist of (a combination of) custom-compiled code, declarative queries, and/or expensive machine learning (ML) techniques. Examples include sentiment analysis [22] of social media posts, named entity extraction [14], face recognition [32, 39], and missing value imputation in relational data [33].

Data enrichment could be performed as a *periodic offline process* prior to loading the data into a database for analysis. For instance, in the enterprise data warehouses, the data collected from diverse transactional databases is stored in the raw format at the time of data arrival. Such data is periodically loaded into a data warehouse [1–3, 10] after transformation. This strategy can be categorized as an *extract-transform-load* process. This strategy adds a significant delay between the time data arrives (or is created) and when the data is available for analysis. This limits the ability of organizations to analyze data in (near) real-time as it arrives. Systems such as Spark Streaming [58] are capable of executing enrichment functions on the newly arriving data before its storage into a DBMS. Recently, [52] has explored ways to optimize enrichment during ingestion. However, such systems suffer from several limitations.

Limitations of data enrichment at ingestion. Enriching data at arrival exhibits several limitations: (*i*) *Unnecessary enrichment*: Enriching data at ingestion could incur high overhead of redundant data enrichment, if the analyst uses only (a small) portion of the data. If workloads are predictable, one could potentially limit enrichment to only data that is expected to be used. However, accurate prediction of the workload can be difficult, as argued in [24, 46]. (*ii*) *No support of complex enrichment:* Enriching data at ingestion is only feasible if enrichment functions are not computationally expensive. When functions use complex ML models (*e.g.*, Multi-layer Perceptron, Random Forest), executing them at ingestion would create a bottleneck. (*iii*) *Limited data ingestion:* Enriching all data as it arrives limits the system to ingest only 10s of events per second.[1]

In order to reduce latency between the time when data arrives/is created and when data is available for analysis that *require data to be enriched before using, **this paper explores effective ways to integrate data enrichment into query processing in databases***.

Challenges of data enrichment at query execution. Integrating data enrichment with query processing raises several challenges: (*i*) How to determine which data items/objects need to be enriched to answer a query correctly. (*ii*) Where should enrichments be performed — either closer to the data at the DBMS possibly using stored procedures and user-defined functions (UDFs), or outside of the DBMS in an enrichment server. Both offer different pros and cons in terms of data movement, redundant enrichment, and scope for parallelism. (*iii*) How to reduce the query execution time — while enrichment at query time reduces the amount of work at ingestion (hence, scaling ingestion to higher data rates), it potentially causes an increased query execution time for individual queries.

Our contributions. This paper addresses the above-mentioned challenges. Our contributions in the paper are as follows:

(1) We develop and discuss pros/cons of two distinct solutions to support joint query processing and data enrichment: (*i*) a ***loosely coupled*** approach (referred to as *loose* design) that performs data enrichment at an external server (different from a database server), called as *enrichment server* and (*ii*) a ***tightly coupled*** approach (referred to as *tight* design) that uses stored procedures and UDFs to co-process enrichment and queries at the DBMS. Both strategies attempt to minimize the number of redundant enrichments.
(2) We address the challenge of increased query latency due to enrichment using a progressive approach of answering queries. In particular, we implement efficient strategies that use incremental view maintenance (IVM), UDFs, and their batched execution to enrich and maintain query results efficiently during query execution.
(3) We experimentally evaluate both the designs in various domains of social media and multimedia data using various enrichment functions. Results show that both the designs outperform significantly than the approach of enriching data at ingestion.

In [12] we envisioned a data management system that supports enrichment transparently to the end-users and identified several challenges. We defined a new data model and presented application scenarios for the proposed system. This paper addresses the implementation challenges identified in [12] and implements a tightly-coupled and loosely-coupled data management system that supports enrichment of data at query time. Furthermore, in [8] we

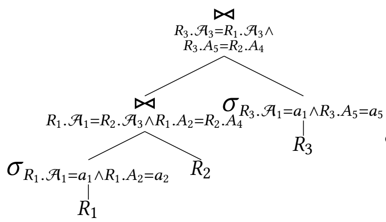
[1]The challenge of executing complex ML functions on data as it arrives, was discussed extensively in the curated session of SIGMOD 2021 [40], that identified that often organizations are forced to use simpler functions that can be performed at ingestion, even though it results in poor quality.

tid	UserID	Tweet	feature	location	TweetTime	**topic**	**sentiment**
t_1	John	Uploading pics on Facebook.	[0.2, ..., 0.4]	US	16:08	social media	positive
t_2	Mark	Feeling great and listening to music.	[0.5, ..., 0.3]	US	16:48	entertainment	NULL
t_3	Richard	Sad about current pandemic.	[0.6, ..., 0.4]	UK	11:48	NULL	NULL

Table 1: `TweetData` table where `topic` and `sentiment` are the derived attributes.

SELECT * FROM R_1, R_2, R_3
WHERE $R_1.\mathcal{A}_1 = a_1$ AND
$R_1.A_2 = a_2$ AND $R_1.\mathcal{A}_1 = R_2.\mathcal{A}_3$
AND $R_1.A_2 = R_2.A_4$ AND
$R_3.\mathcal{A}_1 = a_1$ AND $R_3.A_5 = a_5$ AND
$R_3.\mathcal{A}_3 = R_1.\mathcal{A}_3$ AND
$R_3.A_5 = R_2.A_4$

(a) Original query.



(b) Query tree.

Figure 1: Original query and its query tree.

developed approaches to order enrichments that optimize progressiveness of queries using the loose design proposed in this paper.

2 ENRICHMENT AT QUERY PROCESSING

Before presenting our approaches for data enrichment during query execution, we discuss our data model that is an extended relational model and the notion of enrichment functions. In our data model, some attributes of a relation are ***derived*** (denoted as $\mathcal{A}_i$) and require enrichment; the remaining attributes are ***fixed*** (denoted as A_j) and do not require enrichment. Enrichment is performed by a set of associated enrichment functions with $\mathcal{A}_i$. Without loss of generality, all relations contain an `id` attribute to uniquely identify tuples. *E.g.*, in a relation storing tweets, a derived attribute can be the tweet's `sentiment`, which is enriched using sentiment analysis functions on the tweet. Likewise, in a relation storing images, the identity of people in images can be the derived attribute, which is enriched using face recognition techniques for identifying a person.

In general, several *enrichment functions* could be used either independently or in combination to determine the value of a derived attribute. If an enrichment function is executed on a tuple, the derived attribute will take the value of the function output. *If there are no **enrichment function executed so far**, then the **attribute value will be** **NULL***. For example, in Table 1, the values of two derived attributes `topic` and `sentiment` are NULL in t_3, since they are not enriched yet.

2.1 Query Processing in Loose Design

This design executes enrichment outside of the DBMS at an enrichment server. Given a query q, the key step is to generate `probe queries` ($pq(R_i)$) — for each relation R_i whose derived attributed is a part of q — to identify a "minimal" subset of tuples (as small a subset as possible) that need to be enriched to execute q. The retrieved tuples by `probe queries` are enriched in the enrichment server, and the corresponding modified (enriched) values are updated at the DBMS. Finally, query q is executed at the DBMS.

It exploits the following strategies to identify the minimal subset:

- *Exploiting Prior Work:* $pq(R_i)$ filters out all tuples of R_i that have been enriched earlier (*e.g.*, as part of prior queries), and hence, their derived attribute values in the database are not NULL.
- *Exploiting Selection Conditions on Fixed Attributes:* $pq(R_i)$ filters all tuples of R_i that do not satisfy selection conditions over fixed attributes of R_i. *E.g.*, for the query of Figure 1a, to identify the tuples of R_1 that require enrichment, $pq(R_1)$ retrieves only those tuples of R_1 that satisfy the condition $R_1.A_2 = a_2$.
- *Exploiting Join Conditions on Fixed Attributes:* $pq(R_i)$ filters out all tuples of R_i that would not join with any tuples in R_j, if a join condition exists between R_i and R_j in q based on fixed attributes. *E.g.*, for the query of Figure 1a, the tuples of relation R_1 that do not match with any tuples of R_3 based on the join condition of $R_3.A_1 = R_1.A_1$ do not need to be enriched.

Probe Query Generation Steps. The steps for generating `probe queries` (based on above three strategies) are as follows:

[Step 0]: *Query Tree Generation*: An input query q is first converted into a corresponding query tree, in which, selection conditions are pushed down as much as possible. The conditions present in selection and join nodes are converted into a conjunctive normal form (CNF), *i.e.*, ($C = C_1 \wedge C_2 \wedge \ldots \wedge C_z$). Each condition $C_i \in C$ is characterized as either a ***fixed condition*** (*i.e.*, a condition containing only fixed attributes) or a ***derived condition*** (*i.e.*, a condition containing only derived or both fixed and derived attributes).

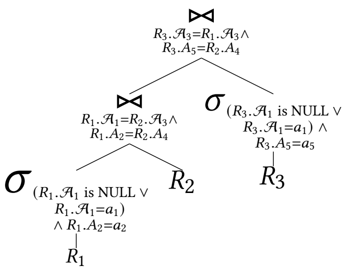
Figure 1b shows the query tree generated from the query of Figure 1a. In a CNF condition $R_1.\mathcal{A}_1 = a_1 \wedge R_1.A_2 = a_2$, the condition $R_1.A_2 = a_2$ is a fixed condition, while $R_1.\mathcal{A}_1 = a_1$ is derived.

[Step 1]: *Rewrite of Selection Condition ($\sigma_C(R)$)*: Given a CNF condition C at a selection node, for each derived condition $C_i \in C$ over derived attribute(s) $\mathcal{A}_1, \ldots, \mathcal{A}_n$, this step finds only those tuples for which there exists an attribute $\mathcal{A}_{i\in[1,\ldots,n]}$ that has not been enriched before. This filtering is achieved by replacing C_i by $\left[(\bigvee_{i=1}^{n} \mathcal{A}_i = \text{NULL}) \vee C_i\right]$. The fixed conditions are kept identical.

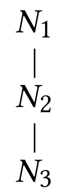
Figure 2a, for the CNF expression $R_1.\mathcal{A}_1 = a_1 \wedge R_1.A_2 = a_2$ as shown in Figure 1b, shows rewritten selection as: $((R_1.\mathcal{A}_1 \text{ is NULL} \vee R_1.\mathcal{A}_1 = a_1) \wedge R_1.A_2 = a_2)$. Note that only the first condition is modified as it is a derived condition, while the second condition is kept identical as it is fixed.

[Step 2]: *Generating Join Graph*: This step and the next step 3 exploits join conditions on fixed attributes in a query to filter out tuples of R_i that do not require enrichment. Given a modified query tree using Step 1 for selection conditions, now, a ***join-graph*** is generated from the tree. The purpose of the join graph is to find out for a relation R_i in the query: which join conditions (on fixed attribute) with other relations can be utilized to reduce the number of tuples of R_i that require enrichment.

In the join graph, the nodes correspond to *reduced* relations, *i.e.*, relations with the selection conditions applied on them. An edge between two nodes shows the join conditions between two relations (based on the original query) expressed in CNF form. Next, from each edge of the join graph, all the derived join conditions are removed. If all the conjuncts are on derived attributes, then we obtain just graph nodes that show none of the join conditions



(a) Step 1: Rewritten query tree for selection conditions.



(b) Step 2: Join graph (Node N_1 : $\sigma_{(R_1.\mathcal{A}_1 \text{ is NULL} \vee R_1.\mathcal{A}_1=a_1) \wedge R_1.A_2=a_2}(R_1)$, Node N_2 : R_2, and Node N_3 : $\sigma_{(R_3.\mathcal{A}_1 \text{ is NULL} \vee R_3.\mathcal{A}_1=a_1) \wedge R_3.A_5=a_5}(R_3)$).

$$\langle N_2 \ltimes_{R_2.A_4=R_3.A_5} N_3; \\ N_1 \ltimes_{R_1.A_2=R_2.A_4} N_2; \rangle$$

(c) Step 3: Semi-join programs for R_1.

$$\sigma_{R_1.A_2=a_2 \wedge (R_1.\mathcal{A}_1 \text{ is NULL} \vee R_1.\mathcal{A}_1=a_1)}(R_1) \ltimes_{R_1.A_2=R_2.A_4} \left[R_2 \ltimes_{R_2.A_4=R_3.A_5} (\sigma_{(R_3.A_5=a_5) \wedge (R_3.\mathcal{A}_1 \text{ is NULL} \vee R_3.\mathcal{A}_1=a_1)}(R_3))\right]$$

(d) Step 4: The `probe query` for R_1.

Figure 2: Steps involved in loose design.

between the two relations can be exploited to reduce the set of tuples that require enrichment. (In a query tree `union`, `set-difference`, or `cross product` operators are ignored, since they cannot be utilized to reduce the number of tuples in `probe queries` apart from the join conditions.)

Figure 2b shows a join-graph for the query tree of Figure 2a. This graph contains three nodes: $\langle N_1, N_2, N_3 \rangle$, representing the reduced relations of $\langle R_1, R_2, R_3 \rangle$, respectively, *i.e.*, after applying selection conditions on each relation. Edge between N_1 and N_2 represents the join condition $R_1.A_2 = R_2.A_4$ (after removing the join condition $R_1.\mathcal{A}_1 = R_2.\mathcal{A}_3$ on derived attributes from Figure 2a).

[Step 3]: ***Semi-join Program Generation***: Given the join graph as an input, for each node N_i in the graph, this step generates a set of semi-join programs for N_i to reduce the number of tuples of N_i that require enrichment. For N_i, semi-join programs are generated by exploiting join conditions among nodes of the graph. For node N_i, this step starts from node N_i in the join graph and generates a spanning tree, denoted as $ST(N_i)$, that contains all nodes of the graph with the minimum possible number of edges (using breadth-first traversal). From $ST(N_i)$, multiple semi-join programs are generated based on the join conditions in $ST(N_i)$.

Semi-join programs for a node N_i are generated in a bottom-up manner from $ST(N_i)$ starting from the children nodes and reaching upto N_i. For each node encountered in the path, a semi-join program is generated. The nodes in $ST(N_i)$ are traversed in a breadth-first order from the leaf node to the root node. All the semi-join programs between the leaf node and their immediate parent nodes are created first. This step is continued until all the paths from the leaf node to the root node are consumed.

For example, $ST(N_1)$ for node N_1, is a tree with root as the node N_1, the node N_2 as the child of N_1, and the node N_3 as the child of N_2 (same as the graph shown in Figure 2b). In $ST(N_1)$ (Figure 2b), a semi-join between relations N_2 and N_3 is performed first to identify the tuples of relation R_2 (part of N_2) that may result in the join output of R_2 and R_3 (as shown in Figure 2c). After this, a semi-join between R_1 and the tuples of R_2 output from the previous semi-join, is performed. Using these two semi-join programs, this step is able to eliminate two types of tuples from R_1: (*i*) the tuples of R_1 that do not join with any tuple of R_2 and R_3, and (*ii*) the tuples of R_1 that may join with some tuples of R_2 but ultimately do not join with any tuple of R_3. This step for semi-join reduction we used is based on the seminal work on semi-join given in [18].

[Step 4]: ***Generating*** **`probe queries`**: Given the semi-join programs (obtained in the previous step), this step generates a `probe query` based on the semi-join programs and the selection conditions on R_i in a straightforward manner. For example, in Figure 2d, we show the `probe query` generated for R_1, from the semi-join programs described in Figure 2c and the selection conditions added to the query tree of Figure 2a for R_1.

2.2 Query Processing in Tight Design

The tight design rewrites an input query q into a modified query q' that checks whether each derived attribute $\mathcal{A}_i \in q$ has been enriched earlier. If not, q' invokes a UDF, called $read_u$ UDF that executes enrichment functions and updates the value of $\mathcal{A}_i$.The $read_u$ is implemented as a generic function that takes as inputs: the name of the relation (*e.g.*, 'R_i'), the name of the derived attribute (*e.g.*, '$\mathcal{A}_j$'), tuple identity, and the identity of an enrichment function.

Rewrite of Selection Condition: This design rewrites each selection condition $(R.\mathcal{A}_i \ op\ a_i) \in q$ (where op is $\geq$, $>$, $=$, $\leq$, $<$, or $\neq$, and a_i is a constant value) that contains a derived attribute, by a modified selection condition denoted as $\omega_\sigma(R.\mathcal{A}_i \ op\ a_i)$, as follows:

$$R.\mathcal{A}_i \ op\ a_i \ \vee [R.\mathcal{A}_i \text{ is NULL} \wedge read_u(\text{'}R\text{'}, \text{'}\mathcal{A}_i\text{'}, R.id, f_{\mathcal{A}_i}.id) \ op\ a_i]$$

Here, $f_{\mathcal{A}_i}.id$ refers to the identity of an enrichment function for $\mathcal{A}_i$. In this rewritten condition, if a tuple's value in $\mathcal{A}_i$ is already enriched, then the original selection condition is evaluated (*i.e.*, $R.\mathcal{A}_i \ op\ a_i$). Otherwise, $read_u$ UDF is executed on the tuple to enrich the attribute $\mathcal{A}_i$ first, and then the selection condition is executed. Note that $read_u$ UDF is only invoked if the attribute value has not been enriched before.

Rewrite of Join Condition: We rewrite each join condition $R_p.\mathcal{A}_i \ op\ R_q.\mathcal{A}_j \in q$ that contains derived attributes $\mathcal{A}_i$ and $\mathcal{A}_j$, by a modified join condition, denoted as $\omega_\sigma(R.\mathcal{A}_i \ op\ a_i)$, based on whether one (or both) of the derived attributes in the condition have previously been enriched. If both the derived attributes have been enriched, $(R_p.\mathcal{A}_i \ op R_q.\mathcal{A}_j)$ is executed with no modification. If one of the attributes (say $R_p.\mathcal{A}_i$) is not enriched, then $R_p.\mathcal{A}_i$ is replaced with a call to UDF $read_u$ on $R_p.\mathcal{A}_i$ in order to enrich the attribute as part of checking the join condition. If both of the attributes (*i.e.*, $R_p.\mathcal{A}_i$ and $R_q.\mathcal{A}_j$) are not enriched, then both attributes in the join condition are replaced by calls to the $read_u$ UDF. The modified join condition of $\omega_{\bowtie}(R_p.\mathcal{A}_i \ op\ R_q.\mathcal{A}_j)$ is shown below:

$R_p.\mathcal{A}_i \; op \; R_q.\mathcal{A}_j$ /*Both $\mathcal{A}_i$ and $\mathcal{A}_j$ are enriched*/

$\lor \big[R_p.\mathcal{A}_i$ is not NULL $\land R_q.\mathcal{A}_j$ is NULL /* Only $\mathcal{A}_i$ is enriched */

$\land \; read_u(`R_q', `\mathcal{A}_j', R_q.id, f_{\mathcal{A}_j}.id,) \; op \; R_p.\mathcal{A}_i\big]$

$\lor \big[R_p.\mathcal{A}_i$ is NULL $\land R_q.\mathcal{A}_j$ is not NULL /*Only $\mathcal{A}_j$ is enriched*/

$\land \; read_u(`R_p', `\mathcal{A}_i', R_p.id, f_{\mathcal{A}_i}.id) \; op \; R_q.\mathcal{A}_j\big]$

$\lor \big[R_p.\mathcal{A}_i$ is NULL $\land R_q.\mathcal{A}_j$ is NULL /* None of $\mathcal{A}_i$ or $\mathcal{A}_j$ are enriched*/

$\land \; read_u(`R_p', `\mathcal{A}_i', R_p.id, f_{\mathcal{A}_i}.id) \; op \; read_u(`R_q', `\mathcal{A}_j', R_q.id, f_{\mathcal{A}_j}.id)\big]$

Example. Below, we illustrate rewritten queries for the query of Figure 1a using modified selection (ω_σ) and join conditions ($\omega_{\bowtie}$) as described above.

SELECT * FROM R_1, R_2, R_3 WHERE $\omega_\sigma(R_1.\mathcal{A}_1 = a_1)$ AND $R_1.A_2 = a_2$ AND $\omega_{\bowtie}(R_1.\mathcal{A}_1 = R_2.\mathcal{A}_3)$ AND $R_1.A_2 = R_2.A_4$ AND $\omega_\sigma(R_3.\mathcal{A}_1 = a_1)$ AND $R_3.A_5 = a_5$ AND $\omega_{\bowtie}(R_3.\mathcal{A}_3 = R_1.\mathcal{A}_3)$ AND $R_3.A_5 = R_2.A_4$

3 PROGRESSIVE QUERY PROCESSING

While both designs reduce redundant enrichment of data and scale to higher ingestion rates (still supporting queries on data as it arrives), they increase query latency due to query time enrichment. To reduce query latency, this section explores ways to make both the design progressive that iteratively refines the query results as more enrichments are performed.

A progressive approach allows data to be consumed by analysts right away and to stop computation at any time they are satisfied with results [15, 43, 54]. Progressive query processing has been extensively explored in the approximate evaluation of aggregation queries (Approximate Query Processing – AQP) [29, 36, 42] to reduce *query latency arising from processing of massive datasets.* In contrast, *in this paper, the challenge is to reduce query latency arising from the execution of expensive enrichment functions.*

To develop a progressive approach, we exploit *a tradeoff between cost and quality*, which ML models often exhibit; cheaper (*i.e.*, low execution cost) functions produce prediction faster with low accuracy, compared to more expensive (*i.e.*, high execution cost) functions that produce slower but high-quality predictions. *E.g.*, a random forest classifier (RF) implemented using a small number of decision tree (DT) models is cheaper but less accurate compared to an RF classifier using many DT models as long as it does not overfit the training data. Such a behavior of classifiers is also highlighted in prior work of [23] that studied cost-accuracy tradeoff of diverse classifiers for sentiment analysis in tweets. Other examples of exploring cost-quality tradeoff of deep neural networks include accelerating performance by reducing floating point precision [25, 49, 59] and ways to reduce the network size by reducing width of the layers, skipping layers, or by skipping modules [20, 53, 56]. Each of these techniques trade complexity (and hence cost) with model precision. Based on this tradeoff, we achieve progressive answering by running cheap functions on (a subset of) data to generate initial answers and subsequently selecting additional data to enrich and/or enrich the old data using more functions to refine the answers.

Below, we develop a progressive approach of enriching data and answering queries for both the loose and tight designs. We begin by defining the semantics of progressive query processing and then describe ways to achieve progressiveness. From here in this section, by loose and tight design we refer to their progressive versions.

3.1 Progressive Queries

This section first describes notations that help in defining teh progressive versions of the loose and tight designs.

Notations. We now permit multiple enrichment functions to be associated with each derived attribute.[2] Suppose $\mathcal{A}$ is a derived attribute and the set of enrichment functions associated with $\mathcal{A}$ is $\{f_1, f_2, \ldots, f_n\}$. This set of enrichment functions is called ***function-family*** of $\mathcal{A}$. For example, the `sentiment` derived attribute of `TweetData` (Table 1) may form a function-family with decision tree (DT), a k-nearest neighbor (KNN), multi-layered perceptron (MLP), or a support vector machines (SVM) classifier.

At any instance of time, for a given tuple t and for a given derived attribute $\mathcal{A}$ of a relation, multiple enrichment functions might have been executed, resulting in the value for $\mathcal{A}$ in t. We refer to the set of functions in the function-family that have executed as the ***state of the derived attribute*** (denoted as $state(t.\mathcal{A})$) in the tuple t. The state of a derived attribute $state(t.\mathcal{A})$ contains two components: ***state-bitmap*** that stores a list of enrichment functions that have been executed on $t.\mathcal{A}$; and ***state-output*** that stores the output of executed enrichment functions on $t.\mathcal{A}$.

Each function-family is associated with a ***determinization function*** that finds the value of $\mathcal{A}$ in t based on $state(t.\mathcal{A})$. The determinization function (denoted by $DET(*)$) could use any ensemble technique [38, 55] for generating a value based on enrichment functions executed so far, *e.g.*, it could use a most likely value, or a value based on majority consensus [38]. We treat the determinization function as a black-box and is independent of the specific function used. Note that $DET(state(t.\mathcal{A}))$ returns a single or a NULL value. NULL value represents a situation when state of the attribute does not provide enough evidence to the determinization function to assign any value for $t.\mathcal{A}$. As more functions execute, the state of $\mathcal{A}$ changes, $DET(state(t.\mathcal{A}))$ computes a new value of $\mathcal{A}$ in t.

The notion of the state of a derived attribute generalizes to the state of tuples, relations, and databases in a straightforward way. The state of a tuple t (or a relation R or a database D) denoted by $state(t)$ (or $state(R_i)$ or $state(D)$) is the concatenation of the state of all derived attributes of t (or the concatenation of the state of all tuples or the concatenation of the state of all relations). Likewise, the concept of determinization also generalizes to a tuple, a relation, and a database, denoted by $DET(state(t))$, $DET(R_i)$, and $DET(D)$,

Progressive Query Processing. Now, we concretely develop the concept of progressive query processing. We discretize the query execution time into *epochs*: $\{e_0, e_1, e_2, \ldots, e_z\}$. e_0 is a special epoch to initialize data structures.[3] In each epoch, we select a set of derived attributes and functions to execute to enrich those attributes.

Let q be a query, and let $R_1, R_2, \ldots, R_n$ be the set of relations that are used in q. Let $state(D, e_k)$ be the resulting state of the database based on all the enrichment functions that have executed in (or before) epoch e_k. Let $DET(state(D, e_k))$ be the corresponding determinized representation of the database, where all derived attributes take a value based on their states.

[2] Progressive approach is still possible when there is a single enrichment function associated with derived attributes since the system can choose a subset of data to enrich progressively. However, it is much more effective when it is able to exploit the tradeoffs between execution time and quality.

[3] For simplicity, we will consider epochs $\{e_1, e_2, \ldots, e_z\}$ to be fixed size in the remainder of the paper, though, the approach does not require this to be the case.

Progressive query execution in an epoch e_k returns the results of query executed over the determinized representation of data, *i.e.*, returning answers to $q(DET(state(D, e_k))$, where the determinized representation of D include outputs of all enrichment functions that have been executed so far. Answers to query q differ from epoch e_{k-1} to e_k, due to the database's state change by enrichment functions in the epoch e_k.

Realizing progressive approach raises two related issues (given below) that we address in this section.

- **Managing State.** State represents the current state of enrichment of all tuples in the database. In other words, the state refers to the information about enrichment functions that have been executed and their outputs. The state helps us to avoid repeated execution of enrichment functions on objects. Since the number of objects and the outputs of enrichment functions can be large (*e.g.*, a probability distribution), efficient ways to represent the state are also needed.
- **Incremental Execution of Enrichment and Queries.** The following two problems are needed to be addressed to execute enrichment and queries in an incremental manner: (*i*) *Selection of objects and enrichment functions:* We need to select a set of ⟨object, enrichment function⟩ pairs that improve the quality of existing query results across different epochs. Sampling-based approaches can be used to select objects/enrichment functions (similar to AQP systems [13, 44]), or a benefit-based approach [26] can be used to optimize specific quality metrics of results (*e.g.*, F_α-measure). (*ii*) *Maintaining query results incrementally to avoid the overhead of computing query results from scratch:* As in each epoch the state of the database changes, a straightforward strategy to compute progressive answers is to simply execute the query at the end of each epoch over the entire determinized representation of the database. Such an approach, however, is wasteful, due to re-executing the query in each epoch without exploiting the work of the previous epochs. Instead, we explore a strategy based on Incremental View Maintenance (IVM) [19, 35, 41] that is supported by several database systems. Such a strategy computes answers as a *delta answer* over the previously reported query answers.

3.2 State Management

In both the loose and tight designs, the state of derived attributes of tuples for a relation R is stored as a separate table, $State(R)$. For each derived attribute, $State(R)$ contains a (state) bitmap and a (state) output vector. As mentioned in §3.1, the bitmap contains a bit for each enrichment function associated with the attribute, where 1 means the function was already executed and 0 means it is yet to execute. The output vector contains the results of the execution of enrichment functions.

In both designs, the state table is maintained in the database. In loose design, since enrichment is performed outside of the database, an in-memory cache for the state table is maintained at the enrichment server to reduce the number of database updates. This cache only contains the tuples that may need to be enriched during the entire query execution (*i.e.*, the result of probe queries, as will be discussed in §3.3), and the updates are pushed to the database at the end of the epoch. State table or cache makes sure that the same derived attribute of a tuple is never enriched using the same enrichment function multiple times in both approaches.

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

Table 2: `TweetDataState` table (created for `TweetData` table).

Example. Table 2 shows a state table for `TweetData` table (see Table 1) for `topic` and `sentiment` derived attributes. Consider tuple t_2 bitmap for `sentiment` attribute; that shows enrichment functions 1 and 3 were executed while function 2 is not yet executed. Enrichment functions 1 and 2 are probabilistic classifiers and their outputs were probability distributions $[0.2, 0.6, 0.2, \dots]$ and $[0.86, 0.1, 0.04,]$, respectively, over an ordered domain of values. ■

Compressed State Representation. In the case of a large domain size of a derived attribute, the columns corresponding to its state output can be large. *E.g.*, if domain size of `topic` in `TweetData` is 40 and there are 3 enrichment functions, then `TopicStateOutput` column (see Table 2) could contain 120 values in each row. Such a large domain could incur high storage overhead and read/write cost of the states. Instead, both designs use a compressed representation for state output when domain sizes are large. It sets a ***cutoff threshold*** and only stores the domain values whose probability is above that threshold. Domain values are appropriately mapped to integers using a dictionary encoding and the probabilities are stored as key-value pairs. The compressed representation does not store large tails of a probability distribution.[4]

3.3 Joint Enrichment and Query Execution

Both designs perform enrichment in the epochs and require an update of the query results at the end of epochs. Instead of re-executing the query to find the modified answers, we use an incremental query processing approach based on Incremental View Maintenance (IVM). Below, we discuss how IVM supports incremental processing, and how it is integrated in both the designs.

Background on Incremental View Maintenance (IVM). Given a view corresponding to a query q, for each table $R_i \in q$, IVM algebraically derives an incremental query Δq that is executed (*e.g.*, using triggers as in [35]) whenever the base tables change. Δq query computes only the delta changes of the materialized view q. Correctness of IVM is characterized by ensuring that: $[q(D+\Delta D) = q(D) + \Delta q(D, \Delta D)]$, where D is an instantiation of a database, ΔD are the updates to D, $q(D)$ is the prior query results based on D, Δq is the modified query that needs to be executed on ΔD, and the notation '+' in the expression $q(D) + \Delta q(D, \Delta D)$ refers to the way of combining answers of the two queries to generate the overall answer to q over the modified data.

[19, 35, 41] provide a comprehensive description of how Δq can be algebraically derived from q. Below, to provide intuition, we provide examples of how operators are transformed. Let ΔR_1 and ΔR_2 be the set of tuples updated to relations R_1 and R_2, respectively.

[4]Though, at times, it may require re-execution of enrichment functions, if the determinization process requires a probability value from the corresponding enrichment function for the domain value that has been pruned out. [34] uses a similar strategy compressed representation to store some tuples (with probability higher than a threshold) of a relation in a faster primary index, called *Uncertain Primary Indexing* (UPI), and the remaining tuples in a slower secondary index.

- Let $q = \sigma_C(R_1)$, where C represents a set of selection conditions, then $\Delta q = \sigma_C(\Delta R_1)$, *i.e.*, the selection condition needs to be applied only on the updated tuples of R_1.
- Let $q = R_1 \bowtie R_2$, then $\Delta q = (\Delta R_1 \bowtie R_2 + R_1 \bowtie \Delta R_2 + \Delta R_1 \bowtie \Delta R_2)$, *i.e.*, the updated tuples of R_1 needs to be joined with R_2, the updated tuples of R_2 with R_1 and between the updated tuples of R_1 and R_2.
- Let $q = \gamma_g(R)$, then $\Delta q = \gamma_g(\Delta R_1)$, where γ is an aggregation function, g is a `group by` attribute. The aggregation function γ_g needs to be applied directly on the updated tuples.

Recall that for each of the above queries, the result of the query Δq needs to be merged with the previous results of the query q. Hence, the result of $q(D + \Delta D)$ is obtained by merging the results of $q(D)$ and $\Delta q(D, \Delta D)$. This post-processing step is performed by IVM techniques itself in the DBMS. IVM techniques have been integrated in several popular databases: PostgreSQL [4], Oracle [6], and Amazon Redshift [5]. IVM implementations can be more efficient than recomputing the original query. For example, a rewritten selection query using the above rules requires only selections to be performed on updated tuples (that may be few), compared to re-execute selection over the entire table. Likewise, incremental computation of joins and other operators may be significantly efficient compared to the naïve implementation. DBToaster [35] shows ≈90 times improvement for certain queries in TPC-H benchmark [11], in terms of the number of refreshes supported by IVM, compared to a full refresh of materialized view after each update of base tables.

Incremental Processing. Both designs exploit IVM to incrementally compute modified query answers, as enrichments are performed on the data during epochs. The query execution consists of four steps, discussed in the following subsections. Only one of the four steps (*i.e.*, query setup) is performed once in the zero-th epoch e_0 (this is a special epoch where only query setup is performed), and all other steps are executed iteratively, once per epoch.

3.3.1 *Query Setup*.

During the query setup, both the loose and tight designs initialize a materialized view q_v for the query q based on the current state of the database. Results of q_v are incrementally updated as more data is enriched in future epochs.

In addition, probe queries $pq(R_i)$ (discussed in §2.1) are executed for each relation $R_i \in q$. The query $pq(R_i)$ needs to be modified from §2.1 as simply checking if the value of a derived attribute is not NULL, no longer suffices if a tuple is fully enriched. Instead, the probe queries exploit the state of derived attributes to determine if it can be further enriched. This test is performed by checking if the sum of the bits in the array of $\mathcal{A}j StateBitmap$ column of a tuple is equal to the length of the array in $\mathcal{A}j StateBitmap$ column.

Example 3.1. Considering the probe query of Figure 2d for relation R_1, the modified probe query is shown in Figure 3. In the modified query, if some of the bits in the array of $\mathcal{A}_1 StateBitmap$ column of a tuple is not equal to the length of the array in $\mathcal{A}_1 StateBitmap$ column, then that tuple is not completely enriched and hence it is returned in the probe query result. ■

$$\sigma_{R_1.A_2=a_2 \wedge (array\_sum(\mathcal{A}_1 StateBitmap) != array\_length(\mathcal{A}_1 StateBitmap))}$$
$$(R_1 \bowtie R_1 State) \ltimes_{R_1.A_2=R_2.A_4} \big[R_2 \ltimes_{R_2.A_4=R_3.A_5}$$
$$(\sigma_{(R_3.A_5=a_5) \wedge (array\_sum(\mathcal{A}_1 StateBitmap) != array\_length(\mathcal{A}_1 StateBitmap))}$$
$$(R_3 \bowtie R_3 State)) \big]$$

Figure 3: Updated probe query for R_1.

`PlanSpaceTable`. The result of the probe queries are stored in a table entitled `PlanSpaceTable`. This table stores a set of candidate tuples of relations $R_i \in q$ that are considered for enrichment to answer q. Rows in `PlanSpaceTable` correspond to the name of the relation (R_i) included in q, the tuple ID, and the list of derived attributes for which the tuple needs to be enriched for q (see Table 3).

Rel	TID	Attribute
'R_1'	1	'$\mathcal{A}_1$', '$\mathcal{A}_3$'
	...	
'R_1'	100	'$\mathcal{A}_1$', '$\mathcal{A}_3$'
'R_2'	1	'$\mathcal{A}_2$'
	...	
'R_3'	200	'$\mathcal{A}_1$', '$\mathcal{A}_3$'

Table 3: `PlanSpaceTable`.

Rel	TID	Attr-FID
'R_1'	2	$\langle$'$\mathcal{A}_1$', $f_2.id\rangle$, $\langle$'$\mathcal{A}_3$', $f_5.id\rangle$
'R_1'	3	$\langle$'$\mathcal{A}_1$', $f_4\rangle$, $\langle$'$\mathcal{A}_3$', $f_6.id\rangle$
'R_2'	1	$\langle$'$\mathcal{A}_2$', $f_7.id\rangle$
'R_3'	2	$\langle$'$\mathcal{A}_1$', $f_3.id\rangle$, $\langle$'$\mathcal{A}_3$', $f_5.id\rangle$

Table 4: `PlanTable`.

3.3.2 *Enrichment Planning*.

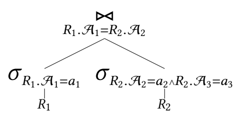
At the beginning of each epoch, based on the state of the tuples, both the loose and tight designs move a set of tuples from `PlanSpaceTable` to a `PlanTable` for (potential) enrichment during this epoch. `PlanTable` contains three columns: `RelationName`, `TID` (tuple identifier), and `Attr-FID` (stores a list of pairs of name of derived attribute and enrichment function identifier), which helps for each tuple and each derived attribute that require enrichment in selecting an enrichment function. A sample `PlanTable` in Table 4 is based on selecting tuples from `PlanSpaceTable` of Table 3. The cost of the selected plan is the summation of the cost of enrichment functions part 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 epoch duration.

In order to populate `PlanTable` from `PlanSpaceTable`, we select a set of ⟨tuple, derived attribute, enrichment function⟩ triplets for enrichment during an epoch. Sample selection methods have been extensively studied for AQP [13, 29, 44]. In such systems, typically a random sample of tuples is selected based on which the approximate aggregate values are computed. Then, such aggregate values are improved as the system progressively chooses a larger sample size and computes the aggregate function on them [29]. Similar to such techniques, we also choose tuples randomly to enrich during a given epoch. However, in contrast to AQP, we need to further select a derived attribute to enrich, as well as, an enrichment function to execute on the chosen attribute (in case more than one enrichment functions are available to enrich). We modify the sampling-based selection policy of AQP for this purpose, leading to three distinct strategies. In each strategy, the tuples to be enriched are chosen randomly from `PlanSpaceTable` by simple random sampling.

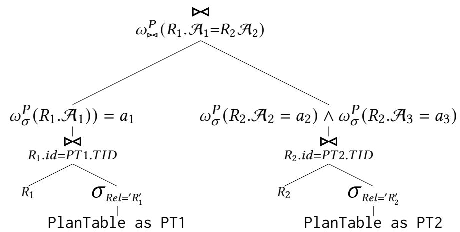
Sampling-based Object Ordered (SB(OO)), where we randomly select a derived attribute from the chosen tuples and enrich it using all the associated functions with the attribute.

Sampling-based Random Ordered (SB(RO)), where we randomly select a derived attribute and randomly select an enrichment function for each of the chosen tuples. The enrichment is continued until the epoch time is exhausted.

Sampling-based Function Ordered (SB(FO)), where we enrich each attribute of the chosen tuples based on an ordered execution of the corresponding functions associated with the attributes. Enrichment functions associated with the attributes are ordered based on their $\frac{quality}{cost}$, where quality is measured using any classifier metrics such as accuracy and cost is measured using the average execution



(a) Original query.



(b) Query used in IVM in the tight design.

Figure 4: The incremental query used by IVM in tight design.

time of the function per object. This strategy is motivated by the optimization of multi-version predicates as proposed in [37].

3.3.3 ***Computing Progressive Answers.*** To compute q progressively, we need to compute delta answers for q_v based on modified data due to enrichment. Computing delta answers in the tight design is more complex than the loose design, which we discuss first.

Progressive Answering in the Loose design. This design performs enrichments at the enrichment server and the modified attribute values of enriched tuples are updated in the tables stored in the DBMS. This update triggers the recomputation of the query answers at the end of each epoch. The IVM query q_v is simply the original query of q. During each epoch e_k, the ⟨tuple, derived attribute, enrichment function⟩ triplets of `PlanTable` are executed, followed by the execution of appropriate determinization function *DET*. The resulting updates are reflected in the database by replacing the current values of the derived attributes of enriched tuples based on the functions executed upto that time. Hence, the determinized representation of the database changes from $DET(state(D, e_{k-1}))$ to $DET(state(D, e_k))$. Such an update, triggers IVM to update the materialized view based on ΔD that consists of all the changes that took place in e_k (*i.e.*, $\big[DET(state(D, e_k))$ - $DET(state(D, e_{k-1}))\big]$). Specifically, $\Delta q(DET(D, e_{k-1}), DET(\Delta D))$ is executed to compute $q(DET(D, e_k))$, *i.e.*, the query result at the end of epoch e_k.

Progressive Answering in the Tight Design. The enrichment of tuples are performed within the query q as part of the UDF execution. Therefore, enrichment of tuples and the subsequent updates to state tables can not be used to trigger the incremental evaluation. To overcome this, the tight design uses the updates to `PlanTable` to trigger the incremental evaluation of query results.

In particular, the query q is rewritten to use `PlanTable` as follows: all relation $R_i \in q$, that require enrichment, R_i is replaced by the expression : $R_i \bowtie_{R_i.TID=PlanTable.TID} (\sigma_{RelName=`R_i'}(PlanTable))$.

Example 3.2. Consider the query of Figure 4a. The tuples of both R_1 and R_2 require enrichment because of the conditions on $\mathcal{A}_1$ and $\mathcal{A}_2$. In the rewritten query of q to support incremental evaluation, both relations are joined with `PlanTable`, as shown in Figure 4b (other rewrites of selection and join conditions are denoted as ω^P_σ and $\omega^P_\bowtie$ will be clear soon). Suppose in e_k, a ⟨tuple, derived attribute, enrichment function⟩ triplet is added to `PlanTable` where the tuple belongs to relation R_1. The addition of this triplet triggers a view update as `PlanTable` is part of the view definition in Figure 4b. ■

In the tight design, each relation R_i is joined with the plan table to determine the set of enrichments to be performed during each epoch. We could potentially reduce the cost of such a join by maintaining plan tables associated with each table separately and joining the relation with its associated plan table. Since plan tables are relatively small (less than 0.1% in size of the database in our experiments), we maintain a single plan table for all the relations.

While the approach to create q_v by rewriting q using `PlanTable` results in desired incremental updates, it suffers from a subtle complexity. Specifically, for a given tuple t, when enrichment functions execute, the change in the state table, results in a new determinized value for the derived attribute $t.\mathcal{A}_j$ in R. If we update the current value of $t.\mathcal{A}_j$ in R, the change would cause the refresh to the view to cascade resulting in a duplicate update to the results. To see this, consider the following example.

Example 3.3. In Example 3.2, suppose a row $\langle t, \mathcal{A}_i, f_j \rangle$ where $t \in R$ is added to `PlanTable` in epoch e_k. Hence, the incremental query execution is triggered. During the execution of incremental query, enrichment function is executed on t and the condition of $(R.\mathcal{A}_1 = a_1)$ is evaluated on it. As a result, the state of t and the determinized value of $t.\mathcal{A}_1$ is updated. Now, if we update the new attribute value of t in R, it will cause another trigger to incremental query execution (as R is part of the q_v), causing duplicate results. ■

To prevent such a situation, we cannot update the value of attribute $t.\mathcal{A}_j$ directly during the execution of q_v. We instead store the value of determinized representation of $t.\mathcal{A}_j$ separately as part of state table R_i`State`. To do so, we extend the schema of R_i`State` to include a new field $\mathcal{A}_j$`Value` for each derived attribute in R_i. R_i`State`, thus, contains three fields: $\mathcal{A}_j$`Bitmap`, $\mathcal{A}_j$`Output`, and $\mathcal{A}_j$`Value`, for attribute $\mathcal{A}_j$.

Since the value of a derived attribute is not modified in place, we need to define additional UDFs for q_v to read the value of $\mathcal{A}_j$. Note that if $\mathcal{A}_j$ is modified (due to enrichment), its value resides in the $\mathcal{A}_j$`Value` column of R_i`State` table. Otherwise, if $\mathcal{A}_j$ is not modified in an epoch, its most recent value is available in $\mathcal{A}_j$ column of table R_i. To enable q_v to correctly retrieve the value of $\mathcal{A}_j$, we define two UDFs of *CheckState* and *GetValue* as described below.

***CheckState* UDF**: This UDF checks if a tuple was enriched for a particular derived attribute present in the query. The input to the UDF is a relation name, a derived attribute name, a tuple identity, and `Attr-FID` value retrieved from `PlanTable` for the corresponding tuple. `Attr-FID` column of `PlanTable`, is used to get the enrichment function that needs to be executed. If that enrichment function was executed before, then it returns true, otherwise it returns false. Note that *CheckState* retrieves this information from the state bitmap column of the derived attribute in the state table §3.2.

***GetValue* UDF:** This UDF retrieves the latest value of a derived attribute for a tuple. It takes as input a relation name (*e.g.*, 'R_i''), attribute name (*e.g.*, '$\mathcal{A}_j$'), and tuple identity and returns the determinized value of $\mathcal{A}_j$ stored in $\mathcal{A}_j$`Value` column of R_i`State` table.

Apart from the above two UDFs, we further need to appropriately modify the $read_u$ function shown in §2.2 (that enriches derived attributes as a side effect of reading them) to account for the way the state and data value are stored when multiple enrichment functions can be associated with a derived attribute.

Modified $read_u$ UDF: Given a tuple, a derived attribute $\mathcal{A}_j$, and an enrichment function, the modified $read_u$ UDF executes the enrichment function on the tuple, updates the state, and returns the determinized representation of the tuple for the derived attribute.

The modified $read_u$ UDF takes the following inputs: the name of the relation, name of the derived attribute, the tuple identity, and the list of ⟨derived attribute, function ID⟩ pairs (stored in *Attr_FID* column of `PlanTable`). It executes the enrichment function on the tuple (parsed from *Attr_FID*), updates the state, and returns the derived attribute value. In state table, it updates the state bitmap, state output, and attribute-value columns. The state-bitmap and state-output are set as the bit and the output of the executed enrichment function. The attribute-value is updated by the latest determinized representation of derived attribute as returned by $read_u$ UDF.

Selection Query. The selection conditions are rewritten to check if the corresponding derived attribute was enriched during the epoch from the state table (enrichment is only performed if the enrichment function was not executed earlier). This is checked by *CheckState* and *GetValue* UDFs as defined earlier. The complete rewrite logic of a selection condition ($\omega_{\sigma}^{P}(R.\mathcal{A}_i \; op \; a_i)$) is presented below:

$$
\begin{aligned}
&\big[CheckState(`R', `\mathcal{A}_i', R.id, Attr\_FID) \quad \text{/* } \mathcal{A}_i \text{ is enriched.*/}\\
&\quad \wedge GetValue(`R', `\mathcal{A}_i', R.id) \; op \; a_i\big]\\
&\vee \big[!CheckState(`R', `\mathcal{A}_i', R.id, Attr\_FID) \quad \text{/* } \mathcal{A}_i \text{ is not enriched.*/}\\
&\quad \wedge read_u(`R', `\mathcal{A}_i', R.id, Attr\_FID) \; op \; a_i\big]
\end{aligned}
$$

The rewritten condition first checks if the tuple is already enriched-for a derived attribute $\mathcal{A}_i$ using *CheckState* UDF. If it is enriched, then the *GetValue* UDF retrieves the attribute value and the selection condition is checked. If a tuple is not enriched before, then $read_u$ UDF is executed on the tuple and the selection condition is checked based on the output of $read_u$ UDF.

Join Query. The rewrite logic for join condition $\omega_{\bowtie}^{P}(R_p.\mathcal{A}_i \; op \; R_q.\mathcal{A}_j))$ is shown below. Given a join condition of $(R_p.\mathcal{A}_i \; op \; R_q.\mathcal{A}_j)$, for a tuple pair, the rewritten condition first checks if both the derived attributes were enriched (*i.e.*, *CheckState* returning true for both tuples). If they were, then the join condition is checked on the output of the *GetValue* function as it returns the latest attribute value of the tuples. In the second and third conditions, only one tuple of the tuple pair was enriched. For the tuple that was enriched before, the value is retrieved using *GetValue* UDF. The other tuple is enriched first using $read_u$ UDF and then the join condition was checked. In the fourth condition, both the tuples were not enriched. Hence, both tuples were enriched using $read_u$ UDF and then join condition was checked.

$$
\begin{aligned}
&\big[CheckState(`R_p', `\mathcal{A}_i', R_p.id, Attr\_FID)\\
&\quad \wedge CheckState(`R_q', `\mathcal{A}_j', R_q.id, Attr\_FID) \quad \text{/* Both of } \mathcal{A}_i \text{ and } \mathcal{A}_j \text{ are enriched*/}\\
&\quad \wedge GetValue(`R_p', `\mathcal{A}_i', R_p.id) \; op \; GetValue(`R_q', `\mathcal{A}_j', R_q.id)\big]\\
&\vee \big[CheckState(`R_p', `\mathcal{A}_i', R_p.id, Attr\_FID)\\
&\quad \wedge\, !\, CheckState(`R_q', `\mathcal{A}_j', R_q.id, Attr\_FID) \quad \text{/* Only } \mathcal{A}_i \text{ is enriched.*/}\\
&\quad \wedge GetValue(`R_p', `\mathcal{A}_i', R_p.id) \; op \; read_u(`R_q', `\mathcal{A}_j', R_q.id, Attr\_FID)\big]\\
&\vee \big[!\, CheckState(`R_p', `\mathcal{A}_i', R_p.id, Attr\_FID)\\
&\quad \wedge\, CheckState(`R_q', `\mathcal{A}_j', R_q.id, Attr\_FID) \quad \text{/* Only } \mathcal{A}_j \text{ is enriched.*/}\\
&\quad \wedge read_u(`R_p', `\mathcal{A}_i', R_p.id, Attr\_FID) \; op \; GetValue(`R_q', `\mathcal{A}_j', R_q.id)\big]\\
&\vee \big[!\, CheckState(`R_p', `\mathcal{A}_i', R_p.id, Attr\_FID)\\
&\quad \wedge\, !\, CheckState(`R_q', `\mathcal{A}_j', R_q.id, Attr\_FID) \quad \text{/* None of } \mathcal{A}_i \text{ and } \mathcal{A}_j \text{ are enriched.*/}\\
&\quad \wedge read_u(`R_p', `\mathcal{A}_i', R_p.id, Attr\_FID) \; op \; read_u(`R_q', `\mathcal{A}_j', R_q.id, Attr\_FID)\big]
\end{aligned}
$$

Example 3.4. Considering the query of Figure 4a, the rewritten query of q_v is shown in Figure 4b. In q_v, the selection and join conditions are rewritten using the rewrite logic of ω_{σ} and $\omega_{\bowtie}$ and contains `PlanTable` as we described above. In an epoch e_k, when a set of ⟨tuple, derived attribute, enrichment function⟩ triplets are added to `PlanTable`, a re-execution of query q_v is triggered. During the execution of q_v, the enrichment of triplets in `PlanTable` take place and the state of the tuples are updated (*i.e.*, using $read_u$ UDF), and IVM maintained for query q_v is updated. Hence, the delta query used to compute the delta changes to the result of q at the end of each epoch e_k is created using the same query of q_v. ■

3.3.4 *Fetching Results*.

In both the loose and tight designs, users can fetch complete query results at the end of an epoch by querying the IVM. If the complete answer set is large, users can retrieve delta changes of answers, *i.e.*, inserted/deleted/updated tuples from the previous epoch. 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 a future version), since the query processing in both designs are not demand-driven, as in SQL databases. The refined answers due to $\Delta q(DET(D, e_{k-1}), DET(\Delta D))$, may result in retraction of previously returned tuples, or addition of new tuples, or updates to the previously reported answers.

3.3.5 *Handling Updates*.

For updates that are not conflicting with any query (*i.e.*, the updated tuples that were not part of the result of the probe queries of the queries that were executing at the time the tuples were updated), we simply reset the state, *i.e.*, set the state-bitmap for that tuple to all zeros. Updates that are conflicting with queries can also be handled by the techniques but require implementation changes that are out of scope for this paper.

3.3.6 *Implementation Details*.

The tight design is implemented using Apache MADlib [30] and the open-source IVM implementation of PostgreSQL [4]. The loose design is implemented using Python and the IVM implementation of PostgreSQL. In tight design, MADlib provides several SQL-like constructs through which users can train ML models on the data stored in the database and use the trained model for inference. We added a new progressive module in the MADlib codebase to implement the tight design. The progressive module provides UDFs that allows users to specify derived attributes, associate enrichment functions (learned using MADlib or other PL/pgSQL and PL/Python UDFs) with the derived attributes. The progressive module implements the UDFs of $read_u$, *CheckState* and *GetValue* UDFs. The queries are wrapped using a stored procedure called *executor* that rewrites the query as an IVM, creates the plan table and plan space table. In each epoch, the executor generates the enrichment plan, populates the plan table, and executes the enrichment functions according to the plan.

4 COMPARISON BETWEEN THE DESIGNS

While loose design enriches tuples prior to query processing (by identifying tuples that might need to be enriched using probe queries), the tight design enriches tuples during query processing when the need to enrich the tuple arises. Tight design may perform lesser number of enrichments since tuples may get eliminated during query processing and, thus, do not need to be enriched.

Relation	#tuples	Size(GB)	Derived attrs.	Functions used
TweetData	11M	10.5	sentiment(3)	GNB,KNN,SVM,MLP
			topic(40)	GNB,KNN,LDA,LR
MultiPie[50]	500K	84.5	gender(2)	DT,GNB,KNN,MLP
			expression(5)	DT, GNB, RF, KNN

Table 5: Datasets used in experiments.

As an example, consider a conjunctive selection query on derived attributes $\mathcal{A}_1$ and $\mathcal{A}_2$. Now, the tight design, after enriching attribute $\mathcal{A}_1$ of a tuple, will not enrich $\mathcal{A}_2$ of the tuple if the tuple does not satisfy the selection condition on $\mathcal{A}_1$. Also, compared to the tight design, the loose design incurs overhead of moving data from DBMS to an enrichment server and vice versa. Such benefits, that are more pronounced when enrichment is expensive, are validated by experiments in §5. Query execution time in the tight design can be further improved by reducing UDF execution cost. Recent research optimized queries with UDFs by executing them on a batch of tuples [52], inlining [47], or executing in parallel. *E.g.*, [47] introduces new algebrization techniques for different types of statements present in a UDF such as DECLARE, SET, IF/ELSE statements. This technique converts a UDF call to a single relational expression that is semantically equivalent to the original UDF. Such optimizations can be applied to optimize the tight design further.

While the loose design incurs overhead due to higher number of enrichment and data movement, it has certain advantages over tight design. The rewritten queries in the tight design are significantly more complex than the loose design, *e.g.*, in the tight design, join conditions in the rewritten query can contain complex disjunctions and UDFs. This makes queries harder to optimize by the standard DBMS optimizers. Different DBMSs provide different optimization supports for UDFs. Systems such as PostgreSQL allows specification of execution cost per tuple of a UDF, which is factored into the plan by the query optimizer. *E.g.*, if a UDF is expensive it may be pulled up in the tree [31]. Disjunctions in the rewritten query may also affect the choice of join algorithm chosen by the optimizer in the query plan. The DBMS may choose to implement the join using a nested loop join while the corresponding join in the plan associated with the original query (which is used in the loose design) may use a hash join instead. Thus, the loose and tight designs offer a tradeoff of reduced enrichment cost and reduced data movement versus possibly complex queries that are harder for existing DBMS to optimize. We study such a tradeoff in the experiments of §5.

5 EXPERIMENTAL EVALUATIONS

This section evaluates the performances of both the loose and tight designs. We address the following questions:

- How much does the tight design save enrichment of tuples compared to the loose design ?
- How does progressive query processing benefit as compared to the approach of completely enriching data before queries ?
- What are the overheads of the designs ?
- How do enrichment plan generation strategies affect progressive query processing? Are there scopes for improvement?

5.1 Experimental Setup

Datasets. We used two datasets: (*i*) TweetData collected using APIs with 11 million rows, two derived attributes: ⟨`sentiment` and `topic`⟩, and six fixed attributes: ⟨`tid`, `UserID`, `Tweet`, `feature`, `location`, and `TweetTime`⟩ (*ii*) MultiPie [50] dataset with 500K facial images, two derived attributes: ⟨`gender` and `expression`⟩, and five fixed attributes: ⟨`ImageID`, `UserID`, `CameraID`, `Image`, and `ImageTime`⟩ (see Table 5).

Q1	SELECT * from MultiPie where gender=1 and CameraID < c_1
Q2	SELECT * from MultiPie where gender = 1 and expression = 2 and CameraID < c_1
Q3	SELECT tid, UserID, Tweet, location, TweetTime from TweetData where sentiment = s_1 and topic = t_1 and TweetTime between(t_1,t_2)
Q4	SELECT * from TweetData T1, TweetData T2 where T1.sentiment = T2.sentiment and T1.topic = T2.topic and T1.TweetTime between(t_1,t_2) and T2.TweetTime between (t_1, t_2)
Q5	SELECT * from MultiPie M1, MultiPie M2 where M1.expression = M2.expression and M1.gender = M2.gender and M1.CameraID < c_1 and M2.CameraID < c_1
Q6	SELECT * from MultiPie M1, MultiPie M2 where M1.gender = M2.gender and M1.expression = 1 and M2.expression = 2 and M1.CameraID < c_1 and M2.CameraID < c_1
Q7	SELECT * from TweetData T1, State S where T1.location = S.city and S.state='California' and T1.sentiment = 1 and T1.TweetTime between(t_1,t_2)
Q8	SELECT * from TweetData T1, TweetData T2, State S where T1.sentiment = T2.sentiment and T1.topic = T2.topic and T1.location = S.city and S.state='California' and T1.TweetTime between(t_1,t_2)
Q9	SELECT topic, count(*) from TweetData where T1.TweetTime between(t_1,t_2) group by sentiment

Table 6: Query templates.

Enrichment Functions. We used the following probabilistic classifiers 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). GNB classifier was calibrated using isotonic-regression model [57], and other classifiers were calibrated using Platt's sigmoid model [45] during cross-validation to output probability distribution.

Queries. Table 6 shows nine queries, where ***Q1-Q3*** are ***selection queries***, ***Q4-Q8*** are ***join queries***, and ***Q9*** is an ***aggregation query***. For experiment 2 we used Q2,Q3, and Q4, and for experiment 6 second part, we used Q3 that contained a derived attribute with a large domain size. The remaining experiments use all the queries.

5.2 Experimental Results

We setup PostgreSQL database on an AWS server with 16 core 2.50 GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD. We used another server with same configuration as the enrichment server in loose design. Enrichment functions were implemented as PostgreSQL UDFs in the tight design and as Python functions in the loose design.

5.2.1 Query time VS Complete Enrichment. To compare the query time approach of both the loose and tight designs against the strategy of complete enrichment before query execution (referred to as ***Baseline***), we used MLP for `sentiment` (100 ms/tweet), GNB for `topic` (125 ms/tweet), MLP for `gender` (1536 ms/image), and RF for `expression` attribute (1380 ms/image). Enriching all 11M tweets of TweetData table for both `topic` and `sentiment` attributes takes ≈**43 hours**[5]. While 43 hours, at first glance, may seem surprisingly

[5]We experimentally measured the runtime of enrichment functions in the tweet dataset for 1 million tweets. The total time taken was 3 hours 55 minutes. Since the dataset

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Baseline	500K	1M	22M	22M	1M	1M	11M	22M	11M
Loose	10K	20K	200K	200K	20K	20K	4K	120K	100K
Tight	10K	13K	164K	127K	12K	12K	4K	114K	100K

Table 7: Exp 1. Number of enrichments in the loose and tight designs, and *Baseline* approach that enriches data completely.

high, we note that prior work (*e.g.*, [23]) has also reported similar run times of inferences on tweets for sentiment analysis. For the Sentiment140 dataset [28] with 1.6 million tweets, the sentiment analysis time is reported as 1 hour 48 minutes which is equivalent to 12.38 hours for 11 million tweets. While [23] only considers a single enrichment – sentiment analysis, if we further perform topic analysis (and that too with a larger number of features than that was used in [23]) it will take ≈43 hours to fully enrich the data. Similarly, complete enrichment of MultiPie data takes ≈**26 hours**.

Exp 1: Number of enrichments. Table 7 shows the number of enrichments in the *Baseline* approach, and the loose and tight designs. Both designs perform significantly better than the *Baseline* approach. The tight design enriches the same or less number of tuples than the loose design it exploits query semantics to avoid redundant enrichments. For queries with multiple predicates (*i.e.*, Q2-Q6 and Q8) on derived attributes, the tuples that did not satisfy a subset of query predicates were not enriched for the remaining derived attributes in the tight design, resulting in the savings in enrichment from the loose design. However, in Q1, Q7, and Q9, the number of enrichments were same in both designs, as Q1 and Q7 had a single predicate on derived attribute and Q9 was an aggregation query with a selection condition on a fixed attribute.

Number of enrichments with varying selectivity. We define selectivity as the ratio of input-cardinality to the output-cardinality of a predicate. We used Q3 where we changed the predicate of `topic` to control the selectivity of probe queries. Table 8 shows that as the predicate selectivity increases (*i.e.*, passes fewer input tuples), the savings in terms of enrichment for both designs increase from the *Baseline* approach. The tight design outperforms the loose design more, when selectivity increases, since all tuples that do not satisfy the predicate of (`topic` $\leq k$) are not further enriched for attribute of `sentiment`. In the loose design, increasing selectivity does not reduce the number of enrichments, as all the tuples that were part of the probe query result are enriched for all derived attributes present in the original query. Thus, for highly selective predicates, the tight design performs better than the loose design, and both designs perform significantly better than *Baseline* for high selectivity values (*i.e.*, 1% and 10%).

Comparison with Eager Enrichment. In this experiment, we ran multiple instances of Q3 one after the other. For each instance of Q3, we randomly picked the value of the time interval predicate such that the query selectivity remains 0.1% of the data, *i.e.*, ≈ 10,000 tweets/query. The time interval is chosen randomly so that the overlap (common tweets) between the query instances remain random. Figure 5 plots the cumulative query execution time of multiple instances of Q3 using query time enrichment and the eager approach of complete enrichment at ingestion. As shown in

had 11 million tweets, we multiplied the run time by a factor of 11. Hence, the total run time of complete enrichment was: 11×(3 hours 55 minutes) ≈ 43 hours.

Approach	Selectivity	`topic` ≤ 10	`topic` ≤ 20	`topic` ≤ 30	`topic` ≤ 40
Baseline	1%	22M	22M	22M	22M
Loose	1%	20K	20K	20K	20K
Tight	1%	11.4K	14.7K	15.4K	16.8K
Baseline	10%	22M	22M	22M	22M
Loose	10%	200K	200K	200K	200K
Tight	10%	100.1K	103.9K	104.6K	139K
Baseline	100%	22M	22M	22M	22M
Loose	100%	22M	22M	22M	22M
Tight	100%	11.16M	12.14M	13.6M	15.8M

Table 8: Exp 1. Number of enrichments performed in the loose and tight designs as compared to *Baseline* with varying selectivity of the fixed condition (*i.e.*, `TweetTime`) in Q3.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Loose	378	684	1291	1319	736	705	32	910	652
Tight	306	572	944	905	627	582	28	924	612

Table 9: Exp 1. Query latency (in seconds).

the figure, the query time approach of enrichment has a very low cumulative execution time compared to eager approach when a lower number of queries covering a small portion of the data are executed. When the number of executed queries are executed become large (*i.e.*, queries cover most of the data), the cumulative cost becomes equal to the cost of eager approach. Hence, the query time approach remains a much better choice than the eager approach over a large number of queries. Even in the scenario where the queries covered all the data, the query time approach performs as good as the enrichment at ingestion approach.

Execution time/Server load. Table 9 shows the latency of queries Q1 to Q9 in the loose and tight designs, where latency is the average execution time of 50 queries generated from each template of Table 6. *E.g.*, we chose 50 queries of *Q*2 by setting different values in the condition on `CameraID` attribute. The latency of queries in both designs are much lower (*i.e.*, two orders of magnitude lower) than the time required to completely enrich the datasets (*i.e.*, 43 hours for tweet datasets and 26 hours for Multipie dataset).

The latency difference between the tight and loose design arise due to three reasons: (*i*) the number of enrichments (shown in Table 7), (*ii*) the data movement cost — transfer of data from DBMS to the enrichment server vice versa (see Table 11), and (*iii*) UDF invocation cost. Table 11 shows the details of time spent in the enrichment server and in the DBMS, and the data transfer cost in the loose design. As expected, the majority of the query execution time is spent at the enrichment server for all queries. In Q2-Q6, the loose design has higher latency due to more enrichments performed by it (see Tables 7 and 11). Furthermore, due to additional data transfer cost, the total time in the loose design is higher than the tight design even for queries that have the same number of enrichments in both designs, *i.e.*, Q1, Q7, and Q9 (see Table 7). Additionally, observe that for Q1, Q7, and Q9, the enrichment costs in the loose design is slightly lower than the tight design since, the enrichment functions are executed in batch in the loose design as compared to the execution of enrichment function UDFs on single rows in the tight design. When we executed the python UDF for sentiment detection on a batch of 1000 tweets, the average execution time

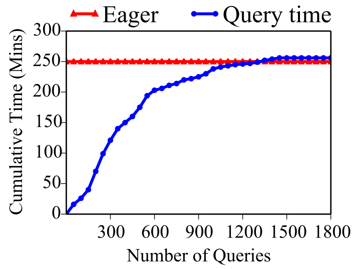


Figure 5: Exp 1. Cumulative time taken by the query time approach as compared to the eager approach for Q3.

of the UDF per tweet was 7.46 ms/tweet. In contrast, the average execution time of PostgreSQL UDF was 7.72 ms/tweet which led to the performance difference.

For Q8, while the number of enrichments performed by both the loose and tight designs are identical (see Table 7), loose design performs better than the tight design. For Q8, in the tight design, the optimizer is not able to optimize the rewritten join condition with UDFs and disjunctions (discussed in §3.3.3) and uses a nested loop join instead of a hash join. However, in the loose design, the optimizer is able to use a hash join since the queries that the loose design runs on the underlying database, does not have any UDFs.

5.2.2 *Progressiveness*.

This section considers all enrichment functions for the derived attributes as shown in Table 5. We present progressive quality improvement of queries Q1-Q9 in two ways: (*i*) plotting the quality of query results with respect to time and (*ii*) quantifying the quality improvement over time using a metric of *progressive score*, denoted by $\mathcal{PS}$. This metric was used in previous literature to measure progressiveness [15, 43].

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

where, $E = \{e_1, e_2, \ldots, e_z\}$ be the epochs, $W(e_i) \in [0, 1]$ is the weight allotted to epoch e_i, $W(e_i) > W(e_{i+1})$, *i.e.*, initial epochs have higher weights, Q is the quality of query answers, and $[Q(Ans(q, e_i)) - Q(Ans(q, e_{i-1}))]$ is the improvement in the quality of answers occurred in the epoch e_i. We chose a linearly decreasing function with a negative slope of 0.05 to assign weights to epochs.

In the remaining experiments, we evaluate the progressive versions of the loose and tight design.

Exp 2: Progressiveness of different queries. Figure 7 evaluates the designs in terms of progressive quality improvement achieved. Figures 7(a), 7(c), and 7(d) show the results for queries Q2, Q3, and Q4 where the quality of answers is measured using ***normalized*** F_1 **measure** *i.e.*, F_1/F_1^{max}, where F_1^{max} is the maximum F_1 measure achieved during query execution by both the designs. Figure 7(b) shows the quality of answers when all enrichment functions used same machine learning algorithm but with different complexities, *i.e.*, random forest classifiers with 5, 10, 15, and 20 base classifiers. The F_1 measure is calculated based on the ground truth data available for the datasets. For aggregation query, the quality is measured using normalized root mean square error (RMSE) from the actual aggregated value calculated from the ground truth. For the aggregation query containing a group-by condition on a derived attribute (*e.g.*, Q9), the RMSE is computed by measuring the deviation for each groups, computing their squares, adding the squares, and finally dividing by the number of groups. We plot normalized measures

State Cutoff	State size (GB)	Progressive Score
0.4	1.4	0.802
0.6	0.9	0.800
0.8	0.7	0.710

Table 10: Exp 5. Effect of state cutoff in state table size and the query performance for Q3.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Loose (DBMS)	3	4	9	11	6	5	2	14	7
Loose (Network)	72	72	37	37	72	72	3	44	37
Loose (ES)	303	608	1245	1271	658	628	27	852	608
Tight (DBMS)	306	572	944	905	627	582	28	924	612

Table 11: Exp 1. Times spent in enrichment server(ES) and DBMS in seconds.

as a function of time to emphasize the rate at which the quality of query results are improved across different queries and datasets, instead of actual F_1-measures. Actual F_1-measure (or RMSE) varies across different queries based on the quality of classifiers chosen for enrichment (*e.g.*, maximum F_1 measures for queries were: Q1 0.73, Q2 0.81, Q3 0.74, Q4 0.89, Q5 0.78, Q6 0.83, Q7 0.82, Q8 0.79, and minimum RMSE for Q9 was 6.58 that was reduced from 132.6).

Figure 7 shows that both the loose and tight designs achieve a high-quality improvement within the first few epochs of query execution. Figure 7(b) highlights that even when the enrichment functions use the same algorithm, both designs are able to achieve progressive improvement in the quality of the query results. The progressive scores achieved for queries (measured by Equation 1) are presented in Figure 6. The tight design achieves a higher progressive score as the number of redundant enrichment is lower than the loose design. This experiment highlights the benefit of progressive query processing that achieves high-quality results within a few epochs, without the need of complete enrichment. The progressive scores in Figure 6 for the loose and tight designs are similar, as the slope set in the progressive score was low (*i.e.*, 0.05). If a steeper function is used, then the difference becomes larger.

Exp 3: Effect of Different Plan Generation Strategies. Figure 8 studies different plan generation strategies (as described in §3.3.2) and their impact on progressiveness. Figure 8 plots progressive improvement of quality for all the queries of Q1 to Q9. Figures 8(a) - 8(i) show that SB(FO) performs the best and SB(OO) performs the worst since SB(FO) chooses functions based on the criteria of $\frac{quality}{cost}$. It allows SB(FO) to select the functions with highest ratio of quality and cost to enrich tuples in the beginning before selecting other functions. In contrast, SB(OO) selects all enrichment functions of a given attribute that results in the enrichment of only a small number of tuples in an epoch. SB(RO) performs marginally better than SB(OO), because of the randomness in the choice of enrichment functions and derived attributes.

5.2.3 *System Overhead*.

We measure the overhead incurred by progressive query processing in both the loose and tight designs.

Exp 4: Time overhead measures the amount of time spent in *non-enrichment tasks*, *i.e.*, query setup, plan selection, delta computation, state update, and UDF invocation to compare against the time involved in data enrichment. The UDF invocation overhead only

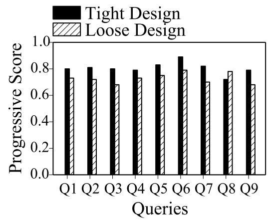


Figure 6: Exp 2. Progressive scores of both designs.

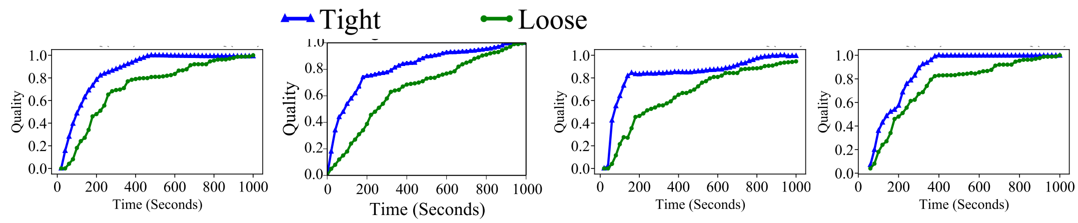


Figure 7: Exp 2. Progressiveness achieved in the loose and tight designs for (a) Q2, (b) Q2 (enrichment functions with same algorithm), (c) Q3, and (d) Q4.

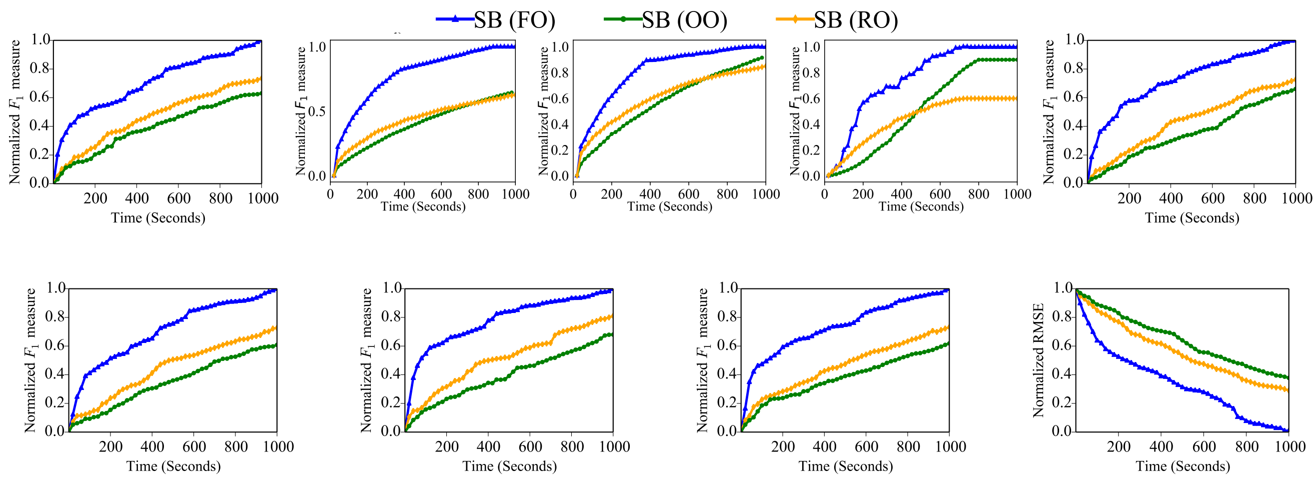


Figure 8: **Exp 3. Comparing different plan generation strategies for all the nine queries: (a) Q1, (b) Q2, (c) Q3, (d) Q4, (e) Q5, (f) Q6, (g) Q7, (h) Q8, (i) Q9.**

exists in tight design as enrichment function UDFs are executed on a single row as compared to the batched execution in the loose design. Particularly, across all epochs, the total time in query setup, plan selection, delta answer computation, state update, and UDF invocation took at most 3s, 4s, 5s, 17s, and 12s respectively, while the total time spent across all epochs in enrichment was 1000s (*i.e.*, almost 3% of the total time was spent in overhead). This result shows that both the loose and tight designs have low overhead of the non-enrichment tasks performed during query processing. Furthermore, we measured the improvement due to the usage of IVM in both designs. Computing the queries from scratch at the end of each epoch resulted in an overhead of 90 seconds for Q7 as compared to the time of 17 seconds due to the usage of IVM.

Exp 5: Storage overhead measures the size of all temporary tables (`PlanSpaceTable` and `PlanTable`), IVM, and the state tables used during query processing to compare against the size of data tables. The maximum storage overheads of `PlanSpaceTable`, `PlanTable`, and IVM at any epoch for the queries of Q1-Q9 were 1.48 MB, 56 KB, and 1.2 MB respectively. The state table sizes for TweetData and Multi-Pie were 2.4 GB and 500 MB, respectively, which are much smaller than the data tables (of 10.5GB and 84.5GB, respectively). Furthermore, using the state cutoff (§3.2) strategy, state storage overhead was reduced significantly. For TweetData, the state overhead was reduced from 2.4 GB (22.9% of the size of TweetData) to 0.9 GB (*i.e.*, 8.6% of the size of TweetData), due to the large domain size (*i.e.*, 40) of `topic` derived attribute.

For `Topic` attribute, we vary the value of cutoff-threshold and examine its effect of it on the query performance for only Q3. Table 10 shows the results. Observe that setting a very high threshold (*i.e.*, 0.8) results in a low storage overhead (*i.e.*, 0.7 GB) of the state table , (but requires re-execution of enrichment functions when queries are posted on the attribute values whose values are not stored in the state table). Hence, it also reduces progressive score (*i.e.*, 0.802 to 0.710) due to more enrichments.

6 RELATED WORK

Our approach of hiding increased query-time latency due to enrichment at query time, by exploiting progressive computation is motivated by AQP systems as discussed in §1 and §3.

Similar works have also been proposed in the past but in a different context of entity resolution [16, 21, 27, 48]. They showed that the query context can be used to eliminate the cleaning of object blocks (residing in the disk) that cannot satisfy the query predicates. [16] utilized an approximate statistic for the objects residing in each disk block. Such statistics are used during query processing to dictate the cleaning tasks. In contrast, we consider a general class of enrichment functions with deterministic as well as probabilistic outputs. We consider state management of tuples that were enriched in the context of previous queries resulting in the elimination of repeated execution of enrichment functions. Furthermore, such frameworks do not consider a progressive approach to query processing when the cost of cleaning functions is high.

Several systems in the past have employed a tightly coupled approach, where application code is pushed down to the DBMS as UDFs [9, 30, 51]. Systems implemented using a loosely coupled approach are also common, where the system was portable to any database system [13, 17, 44].

7 CONCLUSION

In this paper, we proposed a new data management system that supports enrichment during query processing. We explore two different layered architectures for integrating enrichment into query processing: a loosely coupled design where enrichment is performed outside of the DBMS, and a tightly coupled design where enrichment is performed within the DBMS. Both data enrichment strategies come with progressive query processing mechanism. Experimental results on real datasets show the efficacy of both architectures over the naive strategy of complete enrichment and then highlights the tradeoff between the two architectures. When the queries are complex and enrichment costs are the same between the loose and tight designs, the loose design is preferable. In contrast, when the enrichment functions are complex, tight design outperforms due to the saving in enrichment by exploiting query semantics. This paper provides experiments only for single block SPJAG queries. While our approach applies to other types of queries including nested queries, we are restricted by the open-source implementation of Incremental View Materialization (IVM) of the chosen DBMS that only supported single block SPJAG queries. In the future, if the IVM supports nested queries, our implementation will be able to support such queries. Further, the implication of both the loose and tight designs to transactions and mechanisms to leverage enrichment due to concurrent execution of queries are interesting directions of future exploration.

REFERENCES

[1] Amazon redshift data warehouse. https://aws.amazon.com/pm/redshift/.
[2] Google bigquery data warehouse. https://cloud.google.com/bigquery.
[3] Ibm db2 data warehouse. https://www.ibm.com/products/db2-warehouse.
[4] Incremental view maintenance development for postgresql. https://github.com/sraoss/pgsql-ivm.
[5] Incremental view maintenance for amazon redshift. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh-sql-command.html.
[6] Incremental view maintenance for oracle database. https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6002.html.
[7] Internet live stats. http://www.internetlivestats.com.
[8] JENNER: Just-in-time enrichment in query processing https://github.com/dhrubajg246/papers/blob/main/JENNER.pdf.
[9] Sap hana core data services (cds). https://help.sap.com/viewer/09b6623836854766b682356393c6c416/2.0.02/en-US/b710731496cf43b7ba76e15a928f1a80.html.
[10] Snowflake data warehouse. https://www.snowflake.com/workloads/data-warehouse-modernization/.
[11] Transaction processing performance council. tpc-h specification. http://www.tpc.org/tpch/.
[12] A Case for Enrichment in Data Management Systems. *ACM SIGMOD Record*, 2022. https://github.com/dhrubajg246/papers/blob/main/ACaseForEnrichment.pdf.
[13] 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.
[14] S. Agrawal et al. Scalable ad-hoc entity extraction from text collections. *Proc. VLDB Endow.*, 1(1):945–957, Aug. 2008.
[15] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 7(11):999–1010, 2014.
[16] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3):120–131, 2015.
[17] L. Berg, T. Ziegler, C. Binnig, and U. Röhm. Progressivedb: Progressive data analytics as a middleware. *Proc. VLDB Endow.*, 12(12):1814–1817, Aug. 2019.
[18] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[19] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.
[20] S. Braun et al. Towards efficient models for real-time deep noise suppression. In *ICASSP*, pages 656–660. IEEE, 2021.
[21] J. Cambronero, J. K. Feser, M. J. Smith, and S. Madden. Query optimization for dynamic imputation. *Proc. VLDB Endow.*, 10(11):1310–1321, 2017.
[22] 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.
[23] N. C. Dang, M. N. Moreno-García, and F. De la Prieta. Sentiment analysis based on deep learning: A comparative study. *Electronics*, 9(3), 2020.
[24] F. Färber, May, et al. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.
[25] S. Garg, A. Jain, J. Lou, and M. A. Nahmias. Confounding tradeoffs for neural network quantization. *CoRR*, abs/2102.06366, 2021.
[26] D. Ghosh et al. Progressive evaluation of queries over untagged data. *CoRR*, abs/1805.12033, 2018.
[27] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. Cleaning denial constraint violations through relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 805–815, New York, NY, USA, 2020. Association for Computing Machinery.
[28] A. Go, R. Bhayani, and L. Huang. Twitter sentiment classification using distant supervision. 2009.
[29] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.
[30] 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.
[31] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 1993.
[32] 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.
[33] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends Databases*, 5(4):281–393, 2015.
[34] H. Kimura, S. Madden, and S. B. Zdonik. UPI: A primary index for uncertain databases. *Proc. VLDB Endow.*, 3(1):630–637, 2010.
[35] 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.
[36] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. SIGMOD '01, New York, NY, USA, 2001.
[37] I. Lazaridis and S. Mehrotra. Optimization of multi-version expensive predicates. SIGMOD, 2007.
[38] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 1994.
[39] 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.
[40] U. F. Minhas and A. Kumar. SIGMOD 2021 curated session: Data management for ML. https://2021.sigmod.org/program/program_tuesday.shtml.
[41] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264. ACM, 2014.
[42] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow.*, 4(11):1135–1145, 2011.
[43] T. Papenbrock, A. Heise, and F. Naumann. Progressive duplicate detection. *IEEE Trans. Knowl. Data Eng.*, 27(5):1316–1329, 2015.
[44] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *SIGMOD Conference*, pages 1461–1476. ACM, 2018.
[45] 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.
[46] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *Proc. VLDB Endow.*, 7(13):1722–1729, 2014.
[47] K. Ramachandra et al. Froid: Optimization of imperative programs in a relational database. *Proc. VLDB Endow.*, 11(4):432–444, 2017.
[48] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, 2017.
[49] C. D. Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré. High-accuracy low-precision training. *CoRR*, abs/1803.03383, 2018.
[50] T. Sim, S. Baker, and M. Bsat. The CMU pose, illumination, and expression (PIE) database. In *FGR*, pages 53–58. IEEE Computer Society, 2002.

[51] M. Vagac and M. Melichercík. Improving image processing performance using database user-defined functions. In *ICAISC (1)*, volume 9119 of *Lecture Notes in Computer Science*, pages 789–799. Springer, 2015.

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

[53] X. Wang et al. Skipnet: Learning dynamic routing in convolutional networks. In *ECCV (13)*, volume 11217 of *Lecture Notes in Computer Science*, pages 420–436. Springer, 2018.

[54] 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.

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

[56] B. Yang et al. Condconv: Conditionally parameterized convolutions for efficient inference. In *NeurIPS*, pages 1305–1316, 2019.

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

[58] 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.

[59] D. Zhu et al. Efficient precision-adjustable architecture for softmax function in deep learning. *IEEE Trans. Circuits Syst.*, 67-II(12):3382–3386, 2020.