JENNER: Just-in-time Enrichment in Query Processing

ABSTRACT

Several application domains require data to be enriched prior to use. Enrichment, that constitutes interpreting low level data (*e.g.*, low level sensor, text, or image data) into semantically meaningful observations (*e.g.*, person identification, location determination, topic detection), is often performed using expensive machine learning models. Collecting and enriching data offline, prior to loading it to a database, is infeasible if one desires analysis to include recent data. To enable analysis on live data at it arrives, this paper explores a novel approach that jointly enriches and queries the data in order to support interactive exploratory analysis. The technique provides approximate answers that are progressively improved as more data gets enriched. The paper presents a novel strategy JENNER: Just-in-time ENrichmeNt in quERy Processing to select and order objects to enrich in the context of the query in order to improve approximation quality as quickly as possible during query execution. The experimental results on real datasets (images and tweets) show that the approach performs significantly better (~three times better) compared to traditional sampling based approaches.

ACM Reference Format:

. 2021. JENNER: Just-in-time Enrichment in Query Processing . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Today, organizations have access to potentially limitless information in the form of web data repositories, continuously generated sensory data, social media posts, captured audio/video data, click stream data from web portals, and so on [2]. Often, before such data can be analyzed, it needs to be enriched using appropriate machine learning and/or signal processing techniques. Examples of enrichment include sentiment analysis [9, 40] over social media posts, named entity extraction [4, 13] in text, face detection [17] and face recognition [30] in images, and sensor interpretation and fusion [34] over sensory inputs. Functions used to enrich data (which we refer to as *enrichment functions*) often exhibit a cost vs. quality trade-offs where cheaper functions may produce lower quality results whereas expensive functions may result in higher quality.

Limited data enrichment (*e.g.*, by running cheap enrichment functions) at the time of ingestion is typically feasible. However, running a suite of computationally expensive functions to interpret data at ingestion is often impossible due to the speed and volume of the incoming data. Complex ML models (*e.g.*, Multi-layer Perceptron or Random Forest) often take 100s of milliseconds per data item to process on modern processors. If we enrich all data as it arrives, it would limit the system to ingest only 10s of events per second.[1]

Enrichment could also be performed as a separate offline step, but such an approach suffers from several shortcomings. If data is large, it may still be computationally infeasible to fully enrich data. Also, enriching data entirely as part of an offline step can be wasteful, if no query uses the enriched data. Furthermore, in an offline enrichment approach data remains unavailable for analysis until it is enriched. Thus, an offline approach does not allow analysts to interactively analyze new data as it arrives and, finally, it does not support ways to run customized enrichment during analysis.

An alternate is to enrich data during query execution (or during its analysis) when the need for enrichment arises. In such a deferred/lazy approach, since enrichment is performed in the context of a query/analysis task, the specificity of the query can be exploited to only enrich part of the data that is necessary to answer the query. This significantly reduces the overall enrichment cost, especially if analysis is limited to only a small fraction of the entire data, overcoming limitations of the ingest time and offline approaches discussed above.

Performing enrichment at query time adds a new complexity in the form of added latency to query execution. Depending on the query selectivity, such latency can be high. We address this challenge using a combination of two strategies: (*i*) exploit multiple enrichment functions, with a cost/quality trade-off, to perform partial enrichment of objects to provide quick approximate query results, and (*ii*) support a progressive approach to query processing that iteratively refines query results as more data is enriched.

Our goal in this paper is to support interactive analysis on complex unstructured data that may not have been fully enriched either during ingestion time or through an offline pre-processing step. We develop a progessive approach called JENNER : a Just-in-time ENrichmeNt in quERy processing to co-optimize enrichment with query processing. Progressiveness refers to an online mechanism that enriches data in a way that maximizes the rate at which the quality of the answer set of the query improves.

JENNER is well suited for interactive analysis that needs to compute approximate results quickly (without having to wait for the entire data to be fully enriched) and can terminate the query whenever a satisfactory level of quality is achieved. Such a notion of progressiveness has been studied in the literature in various contexts such as in adaptive query processing through sampling [21, 28, 45], in wavelet-based approximate query processing [11, 14], and in data cleaning [7, 29, 35, 39].

The main contribution of this paper is to prioritize/rank which enrichment functions should be evaluated on which objects first to improve the quality of the answer set as quickly as possible, given a query. To this end, JENNER divides the query execution into several

[1]The challenge of running complex ML functions on data as it arrives, was discussed extensively in the curated session of ML in databases in SIGMOD 2021 [31], leading to an observation that often organizations are forced to use simpler functions that can be performed at ingestion, even though such a choice results in poor quality.

tid	User	Tweet	feature	location	retweet_count	like_count	TweetTime	**topic**	**sentiment**
t_1	John	Upload . . .	[0.2, ..., 0.4]	California	40	150	16:08	soc: 0.64	pos: 0.94
t_2	Mark	Listening. . .	[0.5, ..., 0.3]	Chicago	26	100	16:48	soc: 0.4	NULL
t_3	Richard	Iran's . . .	[0.6, ..., 0.4]	New York	80	200	11:48	ent: 0.78	pos:0.7

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

epochs and analyzes the evaluation progress at the beginning of each epoch to generate an execution plan for it. Such a plan consists of multiple *triples* that have the highest potential of improving the quality of the answer in that epoch, where a triple consists of an object, a query predicate, and an enrichment function for evaluating the predicate on the object.

JENNER offers several benefits in the context of interactive analysis. It requires minimal data enrichment at the time of data ingestion (or as a pre-processing step prior to query execution); it only needs to execute a cheap (*i.e.*, with less execution cost) enrichment function that can be used to guide the joint enrichment and query execution. Furthermore, since it merges query execution and data enrichment into a single step, the data is available for analysis immediately after ingestion. Also, it allows analysts to specify and use sophisticated enrichment functions of their choice (that might not have been evaluated at ingestion time).

The deferred/lazy enrichment in JENNER is motivated by prior work on lazy query time data cleaning such as [8]. Such works have developed techniques to combine entity resolution/database repair using denial constraints with query processing to minimize the amount of data that needs to be cleaned. Likewise, [5] developed an approach to dynamically link entities in the context of top-k queries. JENNER, in addition, builds mechanisms to support progressive computation of queries, a problem that has not been studied in the context of query processing in prior work. We believe that the approach developed can also benefit query time data cleaning by supporting progressiveness to it.

In summary, our contributions in this paper are as follows:

- We propose a progressive approach to data enrichment and query evaluation that quantizes the execution time into epochs that are adaptively enriched.
- We present an algorithm for the problem of generating an execution plan that has the highest potential of improving the quality of the answer set in the corresponding epoch.
- We develop an efficient probabilistic strategy to estimate the benefit of applying various enrichment functions on different objects.
- We experimentally evaluate JENNER in different domains (*i.e.*, images and tweets) using real datasets and enrichment functions and demonstrate the efficiency of our solution.

2 DATA MODEL TO SUPPORT ENRICHMENT

We consider an extended relational data model, wherein some of the attributes of a relation are ***derived*** (denoted as $\mathcal{A}_i$) and require enrichment (performed by executing a set of associated enrichment functions). The remaining attributes are ***fixed*** (denoted as A_j) and do not require enrichment. W.l.o.g, all relations contain an `id` attribute that uniquely identifies the tuples present in them.

We will consider a use case of an online tweet analysis application as an illustrative example to explain the notation and the developed techniques. Consider a news organization that collects tweets (using the twitter APIs) relevant to the articles in the front page of their website. They maintain a table entitled **TweetData** shown in Table 1) that consists of several fixed attributes (*e.g.*, user, like_count, retweet_count, TweetTime) and two derived attributes `sentiment` and `topic` derived using sentiment analysis and topic detection functions. Such functions are based on *features* (such as term frequency-inverse document frequency (tf-idf) vector [25] extracted from the tweet text, the collection of prior tweets by the user and/or text associated with the tweets in the reply chain). Several enrichment functions can be associated with both topic and sentiment of increasing complexity that provide a tradeoff between quality and costs. Similar use case examples can be created in domains such as smart space applications where we monitor events and entities using a variety of sensors. For instance, consider an application that analyzes images collected from various surveillance cameras installed in a smart space. They maintain a table of **ImageData** that consists of several fixed attributes of `ImageID`, `CameraID`, `Image`, and `ImageTime` and derived attributes of `person`, `gender`, and `expression` derived using face recognition functions.

The enrichment functions can be categorized based on their input types: (*i*) *single-tuple-input* and (*ii*) *multi-tuple-input*, that take as input a set of fixed attribute values of a single tuple or multiple tuples, respectively. Often classification and regression functions are of type single-tuple-input whereas clustering based functions are of multi-tuple-input type. Enrichment functions can also be categorized based on their output types: (*i*) *single-valued*, (*ii*) *multi-valued*, or (*iii*) *probabilistic*, that output as a prediction a single value (*e.g.*, as in a binary classifier [43]), multiple values (*e.g.*, as in top-k classifiers [23]), or probability distribution over a set of possible values, (as in a *e.g.*, probabilistic classifier). Of the three, probabilistic outputs are the most general since we can always interpret results of the other two as probability distributions. We, thus, assume that the enrichment functions output probabilites in the rest of the paper. For instance, the value of the sentiment attribute in tuple t_1 of `TweetData` in Table 1 is $[0.94, 0.06, 0]$ corresponding to the sentiment values of `positive`, `neutral`, and `negative`. JENNER assumes that the probability outputs of enrichment functions have already been calibrated using mechanisms such as Platt's sigmoid model [37] and isotonic regression model [47] using a labeled dataset. After calibration, output represents a real probability distribution that is learnt during calibration.

The enrichment functions for a derived attribute $\mathcal{A}_i$ are denoted by $F^i = \{f^i_1, f^i_2, \ldots, f^i_k\}$. Each function f^i_j is associated with a *cost* (denoted by c^i_j) which represents the average execution cost of the function on a single tuple. Since the output of an enrichment function is probabilistic, we can associate a notion of uncertainty with the probabilistic outputs. An expensive enrichment function is expected to produce output with less amount of uncertainty. Given a probabilistic attribute value, we can measure uncertainty using the entropy metric [16]. For a tuple t_k and derived attribute $\mathcal{A}_p$,

tid	Topic BitMap	TopicOutput	Topic.Value ([ent, soc, sports, ..])	Sentiment BitMap	Sentiment Output	Sentiment.Value ([pos, neu, neg])
t_1	[1,0,0]	[[0.18,0.64,0.05,...],[],[]]	[0.18, 0.64,...]	[1,0,0]	[[0.94, 0.06,0], [], []]	[0.94, 0.06, 0]
t_2	[1,0,1]	[[0.5,0.2,0.1,...],[],[0.1,0.6,0.1,...]]	[0.3, 0.4,...]	[1,0,1]	[[0.3,0.7,0],[], [0.7,0.3,0]]	[0.5, 0.5, 0]
t_3	[0,1,0]	[[], [0.78,0.06,0.02,...], []]	[0.78, 0.06,...]	[1,1,0]	[[0.1,0.2,0.7], [0.2,0, 0.8],[]]	[0.15, 0.1, 0.75]

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

entropy is calculated as follows:

$$h(t_i, \mathcal{A}_j) = -\sum_i p_i \cdot log(p_i) \quad (1)$$

where, p_i represents the probability of the derived attribute taking the i-th domain value for the tuple t_i. The entropy of t_1 in the example above is, thus, $(-0.94 \times \log(0.94) - 0.06 \times \log(0.06)) = 0.32$.

State and Value of a Derived Attribute. Enrichment state or state of a derived attribute $\mathcal{A}_j$ in tuple t_i (denoted by $state(t_i.\mathcal{A}_j)$) is the information about enrichment functions that have been executed on t_i to derive $\mathcal{A}_j$ and the output of execution of such functions. The state has two components: ***bitmap***, that stores the list of enrichment functions already executed on $t_i.\mathcal{A}_j$; and ***output***, that stores the output of executed enrichment functions on $t_i.\mathcal{A}_j$. As an example, consider that there are three enrichment functions f_1, f_2, and f_3 available for the derived attribute of `sentiment` in Table 2. The state bitmap of t_2 is $\langle 101 \rangle$ signifying that only f_1 and f_3 have been executed on the tuple. Further, the output of the state $\langle [0.3, 0.7, 0], [], [0.7, 0.3, 0] \rangle$ signifies that the output of f_1 was a distribution $[0.3, 0.7, 0]$ and that of f_3 was $[0.7, 0.3, 0]$ over the possible sentiment values of positive, neutral, and negative.

The individual function outputs are aggregated into a combined value denoted by $\mathcal{A}_j.Value$ (*e.g.*, *Sentiment.Value*) using a **combiner function** (*e.g.*, weighted average and majority voting). Since this value depends upon the state of the derived attribute, we denote $\mathcal{A}_j.Value$ by $Val(state(t_i.\mathcal{A}_j))$ and the probability of it taking a a particular value a_j by $Val(state(t_i.\mathcal{A}_j))[a_j]$. For instance, for the example in Table 2, the value of `topic` for t_1, is $Val(state(t_1.\texttt{topic}))$ = [`social media`: 0.64, `entertainment`: 0.18, ...] and the value of `sentiment` is $Val(state(t_1.\texttt{sentiment}))$ = [`positive`: 0.94, `neutral`: 0.06, `negative`: 0]. The probability of t_1 taking the value of `sentiment` as `positive`, *i.e.*, $Val(state(t_1.\texttt{sentiment}))[\texttt{positive}]$ is 0.94.

State and Value of Tuples and Relations. The notions of state and value of derived attributes are generalized to tuples, relations, and the database in a straightforward way. The state (value) of a tuple t_i, denoted by $state(t_i)$ $(Val(state(t_i))$ is the concatenation of the state (value) of all derived attributes of t_i. Likewise, the state (value) of relation R_i and database D is denoted by $state(R_i)$ $(Val(state(R_i)))$ and $state(D)$ $(Val(state(D)))$, respectively.

Next Best Function at a State. Execution of an enrichment function on an attribute $\mathcal{A}_j$ in a tuple t_i in state $state(t_i.\mathcal{A}_j)$ reduces uncertainty in its value $Val(state(t_i.\mathcal{A}_j))$. This reduction in uncertainty depends upon $state(t_i.\mathcal{A}_j)$ and is learnt using a validation data set as a preprocessing step. Given $state(t_i.\mathcal{A}_j)$ we order the enrichment functions associated with $\mathcal{A}_j$ in the order of their uncertainty reduction and choose the one that reduces the uncertainty the most as the next-best function, denoted as $NBF(t_i, \mathcal{A}_j)$). Notice that the uncertainty reduction due to enrichment functions that have already executed in the past is zero and hence they cannot be the next best function at a particular state.

Query Model. JENNER supports single block *select-project-join-aggregation* queries with conditions on both fixed and derived attributes, an example of which is shown in Code Listing 1.

```
SELECT Tweet,location FROM TweetData
WHERE topic="socialMedia" AND sentiment="positive"
AND TweetTime BETWEEN("14:00","16:00");
```

Code Listing 1: Example Query.

If all conditions are on fixed attributes, query result are returned to the user without performing any further enrichment. We thus focus the rest of the paper on queries containing at least one condition on derived attribute for which enrichment may need to be performed during query processing. In the query example above, it contains one condition on a fixed attribute (*i.e.*, on `TweetTime`) and two conditions are on derived attributes (*i.e.*, on `topic` and `sentiment`). While the query above is a set-based query, as will become clear, JENNER supports aggregation queries as well.

Since derived attributes have probabilistic values, JENNER interprets queries based on ***determinization-based*** semantics [26, 33, 46], wherein query Q is evaluated over determinized values for derived attributes in tuples that are part of Q. The determinized value of a derived attribute $\mathcal{A}_j$ in a tuple t_i is determined by executing a determinization function $DET(.)$ on the associated value of the derived attribute (*i.e.*, $DET(Val(state(t_i.\mathcal{A}_j)))$). Several methods to determinize a probabilistic attribute have been previously studied [26, 33, 46]. We choose the function that returns the highest probable value (or NULL if the highest probability value is ambiguous as $DET(.)$. [2]

Progressive Query Execution in Epochs. The complete strategy of progressive query execution is shown in Figure 1. The query execution time is discretized into multiple *epochs* (denoted by $e_1, e_2, \ldots, e_z$) in which data enrichment is performed. We denote the time span of an epoch e_w by the notation $|e_w|$. [3]

During an epoch e_w, certain enrichment functions (that have not been executed so far) are chosen to be executed. Let EP_w =

[2]The techniques developed in the paper can be adopted to other determinization functions studied in [26, 33, 46] including those that return multiple values. This requires specification of a new logic for satisfying conditions present in a query when the attribute values are a list instead of a single value or NULL.

Determinization concept naturally extends to a tuple and a relation. Determinized representation of a relation R is denoted as $DET(R)$:

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

Thus, the execution of query Q is:

$$Q(R_1, R_2, \ldots, R_n) = Q(DET(R_1), DET(R_2), \ldots, DET(R_n))$$

where $DET(R_i)$ is the determinized representation of R_i.

[3]For simplicity, we will consider the duration of each epoch $|e_w|$, $w \in 1, 2, \ldots, n$ to be of fixed size in the remainder of the paper, though, the approach does not require this to be the case.

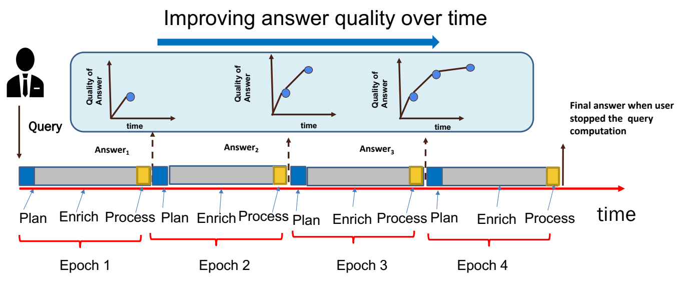


Figure 1: Progressive Query Processing Strategy.

$\{\langle t_i, \mathcal{A}_j, f_k^j\rangle\}$ be such a set containing ⟨tuple, derived attribute, enrichment function⟩ triples. We refer to EP_w as the *enrichment plan* of epoch e_w. Let S be the state of the database at the end of the previous epoch e_{w-1} and, let S' be the state after the execution of enrichment plan EP_w in e_w. This execution results in state update of all the tuples $t_i \in EP_w$ as follows: $\forall\langle t_i, \mathcal{A}_j, f_k^j\rangle \in EP_w$, the $state(t_i.\mathcal{A}_j).bitmap$ is updated by setting the k-th bit to 1 to denote that k-th function is executed. Similarly, $state(t_i.\mathcal{A}_j).output$ is updated to reflect the execution of k-th enrichment function: $state(t_i.\mathcal{A}_j).output = (t_i.\mathcal{A}_j).output \oplus \langle t_i, \mathcal{A}_j, f_k^j\rangle$, where $\oplus$ signifies that the state is updated using k-th array, as well as the update on the derived attribute value of the tuple. At the end of each epoch e_w, the user receives a query result, denoted as Ans_w, based on the current state of the database. Note that a tuple that were part of the answer in previous epoch, may no longer be part of Ans_w. In the rest of the paper, to disambiguate between different states/values of the data during different epochs, we will denote the original database D on which Q executes at D_0 to signify its status at the beginning prior to query being executed. We will refer to the database after the execution of epoch e_w as D_w which will correspond to the database after all the enrichment functions until e_w have executed.

Since in a progressive approach, users may stop query evaluation at any instance of time, performing enrichments that impact the answer quality as early as possible is desirable.

Definition 2.1. Progressive Score. The effectiveness of JENNER is measured using the following progressive score (similar to other progressive approaches [6, 35]):

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

where $E = \{e_1, e_2, \ldots, e_k\}$ is a set of epochs, $W(e_i) \in [0,1]$ is the weight allotted to the epoch e_i, $W(e_i) > W(e_{i-1})$), Qty is the quality of answers, and $[Qty(Ans(Q,e_i)) - Qty(Ans(Q,e_{i-1}))]$ is the improvement in the quality of answers occurred in epoch e_i. ■

Since weights W_i in the progressive score defined above are decreasing, optimizing the score is equivalent to selecting a set of enrichment functions (that have previously not executed) which can result in maximum increase in quality in the following epoch, that is, $Maximize(Qty(Ans(Q,e_i)) - Qty(Ans(Q,e_{i-1})))$.

The quality Qty in Equation 2, for a set-based query answer corresponds to a set-based quality metrics such as precision, recall, F_α-measure [38], or Jaccard similarity coefficient [19]. We define the F_α measure (calculated by the weighted harmonic mean of precision and recall)and Jaccard's similarity below.

$$\begin{aligned} F_\alpha(Ans_w) &= \frac{(1+\alpha)\cdot Pre(Ans_w)\cdot Rec(Ans_w)}{(\alpha\cdot Pre(Ans_w) + Rec(Ans_w))} \\ \mathcal{J}(Ans_w) &= \frac{|Ans_w \cap Ans^{real}|}{|Ans_w \cup Ans^{real}|} = \Big[\frac{1}{Pre(Ans_w)} + \frac{1}{Rec(Ans_w)} - 1\Big] \end{aligned} \tag{3}$$

where Ans^{real} is the real answer of the query in ground truth set G, Pre corresponds to precision, *i.e.*, $Pre(Ans_w) = |Ans_w \cap Ans^{real}|/|Ans_w|$, and Rec corresponds to recall, *i.e.*, $Rec(Ans_w) = |Ans_w \cap Ans^{real}|/|Ans^{real}|$, and $\alpha \in [0,1]$ is the weight factor assigned to precision in calculating F_α-measure. In the rest of the paper, for computing the quality of set-based query result, we restrict our discussion to F_α-measure. The quality of an aggregation query could be measured using root-mean-square error [18] or mean-absolute-error [44].

At each epoch JENNER strives to choose the best set of object to enrich that can improve quality of the query result most. Since the ground truth of objects are not known, JENNER can neither directly measure the quality of the results returned so far. Nor can it precisely determine the improvement in quality by executing an enrichment function. Instead, JENNER estimates both the quality of results (returned during the previous epoch) and the improvement in quality if a selected set of objects are enriched in the current epoch. Based on these estimates, JENNER chooses and executes the enrichments (that maximize the improvement) and returns the resulting answers. Below we discuss how JENNER estimates quality for set-based queries. For aggregation queries, JENNER optimizes the enrichment process using a set-based quality metric and then applies the aggregation function on the set.

Definition 2.2. Estimated Quality Let Ans_w^{MAX} be the set of tuples that have non-zero probability to be in the answer to query Q, Ans_w be a set of tuples returned as an answer to the user. Let $\mathcal{P}_i$ be the probability of a tuple $t_i \in Ans^{real}$ (we discuss ways to compute $\mathcal{P}_i$ later). Furthermore, let m be the cardinality of Ans_w, and n be the cardinality of Ans_w^{MAX}. We can compute estimated precision and recall for Ans_w, denoted $\widehat{Pre}$ and $\widehat{Rec}$ as follows:

$$\widehat{Pre} = \frac{\sum\limits_{t_i\in Ans_w} \mathcal{P}_i}{m}, \quad \widehat{Rec} = \frac{\sum\limits_{t_i\in Ans_w} \mathcal{P}_i}{\sum\limits_{t_j\in Ans_w^{MAX}} \mathcal{P}_j} \tag{4}$$

Given the above estimates of precision and recall, we can next define estimate of F_α measure denoted as $\widehat{F_\alpha}$.

$$\widehat{F_\alpha}(Ans_w) = \frac{(1+\alpha)\sum\limits_{t_i\in Ans_w}\mathcal{P}_i}{\alpha\sum\limits_{t_i\in Ans_w^{MAX}}\mathcal{P}_j + m} \tag{5}$$

The above definition of estimated quality depends upon determining the probability $\mathcal{P}_i$ of an answer tuple t_i to be in the real answer to the query. For a selection query containing a single condition $(t_i.\mathcal{A}_j = a_j)$ on a derived attribute $\mathcal{A}_j$, the $\mathcal{P}_\rangle$ of t_i is simply $Val(state(t_i.\mathcal{A}_j))[a_j]$ if the determinized value of $t_i.\mathcal{A}_j$ corresponds to a_j, else it is zero. For selection conditions on multiple derived attributes, the probability of t_i satisfying the predicate can be computed under the assumption of independence of derived attributes by appropriately combining the probabilities of t_i satisfying the predicates on single derived attributes. For a join query, $\mathcal{P}_i$ is calculated by computing the product of constituting tuples that formed the answer tuple t_i as done in [42] in the context of probabilistic databases.

Progressive Enrichment Problem. Given the notations above, we can now formally state the problem of progressive enrichment. Let Q be a query and let $e_1, e_2, \ldots, e_n$ be the epochs used to execute Q. Let $state(D)$ be the state of the database after the execution of epoch e_{w-1}, where $w \leq n-1$. Let CS_w be the set of tuples in D that are not fully enriched, *i.e.*, for each $t_i \in CS_w$, there exists an enrichment function f_k^j that can be used to enrich a derived attribute $\mathcal{A}_j \in Q$ of t_i, which was not executed before.

The progressive enrichment problem consists of determining a set of ⟨tuple, derived attribute, enrichment function⟩ triples EP_w such that, when executed in epoch e_w, results in a new database value of: $(Val(state(D)) \oplus EP_w)$ that optimizes the following objective function:

$$\begin{aligned} &\max_{\langle t_i, \mathcal{A}_j, f_k^j \rangle} \Big[\widehat{Qty}(Q(DET(Val(state(D)) \oplus EP_w)))- \\ &\widehat{Qty}(Q(Val(state(D))))\Big] \\ &\text{subject to} \\ &\sum_{\langle t_i, \mathcal{A}_j, f_k^j \rangle \in EP_w} cost(\langle t_i, \mathcal{A}_j, f_k^j \rangle) \leq |e_w| \end{aligned} \tag{6}$$

where $\widehat{Qty}(Q(Val(state(D)) \oplus EP_w))$ is the expected quality of the query result when it is executed on the updated state of the database in epoch e_w and $\widehat{Qty}(Q(Val(state(D))))$ is the expected quality of the query result at the end of previous epoch of e_{w-1}.

3 PROGRESSIVE ENRICHMENT IN JENNER

This section describes how JENNER solves the progressive enrichment problem formalized above. We distinguish between the zero-th epoch and the other subsequent epochs in the epoch-based processing of the query. The zero-th epoch performs pre-processing in order for answers to be generated in the subsequent epochs. The subsequent epochs e_w, $w = 1, 2, \ldots, n$ iteratively compute the results. The overall algorithm is depicted in Algorithm 1. We discuss the steps performed during these epochs separately.

Zero-th Epoch (Lines 6 - 13): During the zero-th epoch, as a first step, JENNER identifies for each relation R_i a *minimal set* of candidate tuples whose enrichment in subsequent epochs may influence the query result (denoted as $CandidateSet(R_i)$). Such a $CandidateSet(R_i)$ is identified by executing *probe queries* (Lines 3-5) $pq(R_i)$. Generation of probe queries are discussed in §3.1.

Next, for each t_i in the *CandidateSet* for each relation, for each derived attribute $\mathcal{A}_j$ in t_i that is part of Q, JENNER estimates the probability of t_i matching the condition on $\mathcal{A}_j$ (listed in the code as *match_prob*). For each such attribute $\mathcal{A}_j$ in t_i, JENNER also computes the *benefit* and *cost* of executing the next best function (*NBF*) associated with $t_i.\mathcal{A}_j$ based on its current state (Line 10- 11).

The *match_probability* of a derived attribute $t_i.\mathcal{A}_j$ is determined using the probability $Val(state(t_i.\mathcal{A}_j)).prob[a_j]$ where the selection condition is $\mathcal{A}_j = a_j$. For derived attributes, that do not appear in any selection condition, the value of *match_probability* is 1.

Algorithm 1: Overall Algorithm.

```
Inputs: Query $Q$ and the duration of each epoch $epoch\_duration$.
Outputs: An enrichment plan for each epoch that optimizes
  progressive score.
1  Function $Optimize\_Enrichment()$ begin
2      $CandidateSet^M \leftarrow \emptyset$
       // Beginning of epoch 0.
3      for each $R_i \in Q$ do
4          $pq(R_i) \leftarrow GenerateProbeQuery(Q, R_i)$
5          $CandidateSet(R_i) \leftarrow Execute(pq(R_i))$
6      for each $R_i \in Q$ do
7          for each $t_j \in CandidateSet(R_i)$ do
8              for each $\mathcal{A}_k$ in $R_i$ that is element of $Q$ do
9                  $match\_prob \leftarrow ComputeProb(t_j, \mathcal{A}_k)$
10                 $cost \leftarrow Cost(NBF(t_j, \mathcal{A}_k))$
11                 $benefit \leftarrow ComputeBenefit(NBF(t_j, \mathcal{A}_k))$
12                 $CandidateSet^M \leftarrow CandidateSet^M \cup$
                     $\langle t_j, \mathcal{A}_k, NBF(t_j.\mathcal{A}_k), benefit, cost, match\_prob \rangle$
13     $CandidateSet^M \leftarrow Sort_{match\_prob}(CandidateSet^M)$
       // End of epoch 0 and beginning of epoch w, $w \geq 1$.
14     for each epoch $e_w$ do
15         $EP_w \leftarrow ChooseEnrichmentPlan(CandidateSet^M)$
16         for each entry $\in EP_w$ do
               $ExecuteEnrichment(t, \mathcal{A}, f)$; $UpdateState(t, \mathcal{A}, f)$;
17             $Determinize(t, \mathcal{A})$;
               $UpdateBenefit(CandidateSet^M, t, \mathcal{A}, f)$;
18             // $\langle t, \mathcal{A}, f \rangle$ is the ⟨tuple, derived attribute,
                  enrichment function⟩ of the entry.
19         $Ans_w \leftarrow ProduceQueryResult(Q)$
20     Return $Ans$
```

The cost $Cost(NBF(t_i, \mathcal{A}_j))$ of the next best function for each tuple t_i and the derived attribute $\mathcal{A}_j$ is specified as part of the enrichment function specification as discussed in §2 (Line 10).

The benefit of enrichment of tuple t_i using the next best function (as shown in Line 11) for attribute $\mathcal{A}_j$ is computed by estimating the improvement in quality from the previous epoch. We describe this step in details in §3.2. The benefit, cost, and matching probability for each candidate in $CandidateSet(R_i)$ is stored in the corresponding $CandidateSet^M$ to represent the metadata of candidates, viz. benefit, cost, and probability (Lines 9-12). Candidates in $CandidateSet^M$ are sorted based on their *match_prob* values (Line 13).

Later epochs $e_w, w \geq 1$, (Lines 14 - 19): The later epochs (*i.e.*, $e_1, e_2, \ldots, e_n$) consist of a sequence of the following three steps: (*i*) *Choose Enrichment Plan:* that selects a set of candidate tuples from $CandidateSet^M$ to generate an enrichment plan EP_w for execution during e_w (Line 15); We discuss choosing the enrichment plan in details in §3.2 and in §3.3; (*ii*) *Execute Enrichment Plan:* that enriches the tuples in EP_w, update their state, determinized representations and their benefit in $CandidateSet^M$ (Lines 16 - 17); We discuss them in details in §3.4; (*iii*) *Produce Query Results:* produce a query result by executing the query on determinized representation of the tuples and then choosing a subset of tuples that maximizes the quality of the result measured using $E(F_1)$ measure (Line 19); We discuss this step in details in §3.5. Progressive approach for aggregation queries is realized by developing a progerssive approach for the corresponding set-based query on which the aggregation is performed.

3.1 Probe Query Generation

In order to populate $CandidateSet(R_i)$ for relation R_i, *i.e.*, to find out the set of tuples that may have impact on query results, one could add all tuples that have not been fully enriched to this set. However, such an approach would result in significant number of redundant enrichments. Instead, JENNER exploits the predicates over fixed attributes to filter out tuples whose enrichment has no consequence on the results. For instance, considering a query that contains a selection condition on a fixed attribute, a tuple that does not satisfy the predicate on the attribute could be dropped from considerations for enrichment. Likewise, we can also exploit join predicates on fixed attributes to filter away tuples in a relation that would not satisfy the join predicate. This can be achieved by using a semi-join program as discussed in [10]. We perform filtering of redundant tuples whose enrichment will not affect the query answer by executing a *probe query* for each relation in the original query Q.

The `probe queries` identify a "minimal" subset of tuples (as small a subset as possible) for each $R_i \in Q$ that need to be enriched to execute Q (denoted as $pq(R_i)$). Let us illustrate how probe queries are generated using a sample query in Figure 5a.

To generate a probe query for say relation R_1, we first identify the selection conditions on fixed attributes of R_1. In the query of Figure 5a, this is the condition $R_1.A_2 = a_2$. Thus, we can limit the tuples that require enrichment in R_1 using this condition as shown in Figure 2b. We further exploit join conditions on fixed attributes. For instance, in Figure 5a, an R_1 tuple must join with at-least one of the R_2 tuples that satisfy the restriction on fixed attributes, *i.e.*, $R_2.A_5 = a_5$. A tuple of R_1 will possibly be part of the query answer if it joins some tuples of R_2 which satisfy the join condition $R_1.A_4 = R_2.A_4$. We can determine such a set by computing semi-join with other relations in the query with which R_1 joins using conditions on fixed attributes. Utilizing such a semi-join optimization will result in a nested query as shown in Figure 2c. We restrict the description of probe query generation simply illustrate how it works using an exampe since the algorithm to generate probe queries is a direct adaptation of the technique proposed in [10] for semi-join optimization to reduce the size of relations. We have described the algorithm in Appendix 6.1.

In addition to exploiting selection and join conditions on fixed attributes, we exploit the current state of the tuples to avoid repeating enrichment of tuples that are completely enriched for a derived attribute. This is achieved as shown in Figure 2d by rewriting the selection condition on derived attribute, *i.e.*, $R_1.\mathcal{A}_1 = a_1$ by the condition of: $\big[R_1.id = R_1State.id$ AND $R_1.array\_sum(\mathcal{A}_1 StateBitmap)! = R_1.array\_length(\mathcal{A}_1 StateBitmap)\big]$. This condition checks if a derived attribute of a tuple is completely enriched using the *StateBitmap* column of the R_iState table. For a *StateBitmap* value with all the bits set to 1, the sum of the array and the length of the array becomes equal. All the tuples with bitmap having the sum of elements not equal to the length of the array, are not completely enriched. Hence, such tuples can be part of the probe query result.

SELECT * FROM R_1, R_2 WHERE $R_1.\mathcal{A}_1 = a_1$ AND $R_1.A_2 = a_2$ AND $R_1.\mathcal{A}_3 = R_2.\mathcal{A}_3$ AND $R_1.A_4 = R_2.A_4$ AND $R_2.A_5 = a_5$ AND $R_2.\mathcal{A}_6 = a_6$

(a) Original query.

SELECT * FROM R_1 WHERE $R_1.A_2 = a_2$

(b) Step 1 of probe query generation for relation R_1.

SELECT * FROM R_1 WHERE $R_1.A_2 = a_2$
AND $R_1.A_4$ IN (SELECT A_4 FROM R_2 WHERE $R_2.A_5 = a_5$)

(c) Step 2 of probe query generation for R_1.

SELECT * FROM R_1, R_1State WHERE $R_1.A_2 = a_2$
AND $R_1.A_4$ IN (SELECT A_4 FROM R_2 WHERE $R_2.A_5 = a_5$)
AND $R_1.id = R_1State.id$
AND ($R_1State.array\_sum(\mathcal{A}_1 StateBitmap)! =$
$R_1State.array\_length(\mathcal{A}_1 StateBitmap)$)

(d) Step 3 of probe query generation for R_1.

Figure 2: Steps of `probe query` generation for R_1 in Q.

3.2 Benefit Estimation

JENNER chooses the ⟨tuple, derived attribute, enrichment function⟩ triples from $CandidateSet^M$ as an enrichment plan based on the *benefit* of the triple per unit cost. Benefit, discussed formally below, corresponds to the expected improvement in the quality of the answers compared to the previous epoch because of the execution of the enrichment plan. We restrict the choice of tuples to enrich in the enrichment plan to only those that are not in the answer set of the previous epoch to reduce the overhead of repeatedly computing benefits of $\langle tuple, attribute \rangle$ pairs that have already appeared in the answer. We do so, since the expected benefit of further enriching such a $\langle tuple, attribute \rangle$ pair that already appears in the answer is significantly lower compared to those that are not in the answer. We formally justify this decision using a theorem in Appendix 6.3 and show the performance improvement due to reduced overhead of computing benefit in §4.

Definition 3.1. Benefit of an Enrichment Function. Let Q be a query, D_{w-1} be the database at the end of epoch e_{w-1}, and $\langle t_i, \mathcal{A}_j, f_k \rangle$ be a triple to be executed in epoch e_w. The benefit of $\langle t_i, \mathcal{A}_j, f_k \rangle$ is defined as follows:

$$\begin{aligned} Benefit(\langle t_i, \mathcal{A}_j, f_k \rangle) = \widehat{Qty}(Q(D_{w-1} \oplus \langle t_i, \mathcal{A}_j, f_k \rangle))- \\ \widehat{Qty}(Q(D_{w-1})) \end{aligned} \tag{7}$$

where $\widehat{Qty}(Q(D_{w-1} \oplus \langle t_i, \mathcal{A}_j, f_k \rangle))$ is the estimated quality of the query answer after the triple $\langle t_i, \mathcal{A}_j, f_k \rangle$ is executed and $\widehat{Qty}(Q(D_{w-1}))$ is the estimated quality at the end of e_{w-1}. ■

Thus, to determine the benefit of executing an enrichment function, we need to estimate (*i*) the quality of the answer after epoch e_{w-1} and (*ii*) the expected quality of the answer set if that enrichment function is executed in the current epoch. Before we consider how we can estimate these two metrics for general queries, we first describe how to estimate them for selection queries.

3.2.1 ***Selection Queries***. Given Ans_{w-1} (*i.e.*, $Q(D_{w-1})$), for selection queries, estimating its quality (*i.e.*, $\widehat{Qty}(Q(D_{w-1}))$ is straightforward, since for every tuple $t_i \in Ans_{w-1}$, the probability of the

Algorithm 2: Benefit Calculation.

```
Inputs: A triple containing a tuple $t_i$, a derived attribute $\mathcal{A}_j$, the
  next best function $f_k$ for the tuple at the state of the attribute at
  the end of epoch $e_{w-1}$.
Outputs: The benefit of the ⟨tuple $t_i$, derived attribute $\mathcal{A}_j$,
  enrichment function $f_k$⟩ triplet.
1 Function Compute_Benefit() begin
2     $PreviousQuality \leftarrow \widehat{F}_\alpha(Ans_{w-1})$
3     $\mathcal{E}_{w-1} \leftarrow ComputeEntropy(t_i, \mathcal{A}_j)$
4     $\widehat{\mathcal{E}}_w \leftarrow \mathcal{E}_{w-1} - DeltaEntropy(t_i, \mathcal{A}_j, f_k)$
5     $Match\_Probability \leftarrow ComputeInverseOfEntropy(\widehat{\mathcal{E}}_w)$
6     $ExpectedAnswerQuality \leftarrow$
        $ComputeQuality(Match\_Probability, Ans_{w-1})$
7     $Benefit(t_i, \mathcal{A}_j, f_k) \leftarrow$
        $Max(ExpectedAnswerQuality - PreviousQuality, 0)$
8     Return $Benefit(t_i, \mathcal{A}_j, f_k)$
```

tuple t_i to be in the Ans^{real} is known, as discussed in §2 and illustrated using the following example.

Example 3.1. Consider the selection query in Code Listing 1 on `TweetData` (see Table 1). Suppose at the end of epoch e_1, the state after enrichments that have executed is as shown in Table 2 and let t_1 be part of the query result. Since, t_1 value for `Topic.Value` in Table 2 is [ent: 0.18, social media: 0.64, sports: 0.05] and, thus, its associated determinized value of the topic in Table 1 is `social media`. Likewise, the value in `Sentiment.Value` is [positive:0.94, neutral:0.06, negative: 0], an its determinized representation is `positive`. Thus, the probability of t_1 satisfying the predicate of `sentiment = positive` is 0.94 and `topic = social media` is 0.64. Hence, the combined probability of the tuple satisfying all the selection conditions in the query is $(0.64 \times 0.94) = 0.6$. The expected precision of the answer is 0.6. The recall calculation requires probability of the tuples that are part of the answer as well as of the tuples that are outside of the answer. ■

To compute the benefit for a given $\langle t_i, \mathcal{A}_j, f_k \rangle$, we need to estimate the quality of the answer that would result if Q were to be executed on the database after executing f_k on t_i. Recall that with each enrichment function f_k, we have associated a measure of uncertainty reduction that is a function of the state of the derived attribute $\mathcal{A}_j$ of tuple t_i on which f_k executes. Let the uncertainty reduction for the function f_k executing over the state of the attribute $\mathcal{A}_j$ in tuple t_i in the current database D_{w-1} be Δ. Such an uncertainty reduction measure Δ can allow us to estimate the probability of the tuple satisfying the selection condition of the query after execution of f_k as follows. Let $\mathcal{E}_{w-1}$ be the entropy of attribute $\mathcal{A}_j$ of t_i prior to the execution of f_k. It's entropy after the execution of f_k is thus $\mathcal{E}_{w-1} - \Delta$. We can thus estimate the new probability p of $t_i.\mathcal{A}_j$ satisfying the selection condition by solving the following equation:

$$\mathcal{E}_{w-1} - \Delta = -p \cdot log(p) - (1-p) \cdot log(1-p) \tag{8}$$

Note that the equation above leads to two solutions, one that reduces the probability p of $t_i.\mathcal{A}_j$ satisfying the selection condition (p_{low}) and another that corresponds to the increase in probability (p_{high}).

Example 3.2. Consider tuple t_3 in Table 2 the value of which, based on the execution of the first 2 enrichment functions associated with the `sentiment` attribute, is a distribution [0.15,0.1,0.75] over the possible values of `positive`, neutral, negative. Given the query in Code Listing 1, the probability of t_3 satisfying the predicate of positive is 0.15 and not satisfying the query is 0.85. As a result, entropy is calculated as 0.60. Let us now consider executing the third enrichment function on `sentiment` and, furthermore, let us assume that its associated entropy reduction is 0.3. With the new entropy value of (0.6-0.3) =0.3, we can solve Equation 1 to determine p_{low} and p_{high} as 0.05 and 0.95 respectively. ■

Given (p_{low} and p_{high}) and the probabilities of other tuples satisfying the query condition (which is the same as in the previous epoch e_{w-1}), JENNER can determine the answer that would be return to the user in order to maximize the answer quality (as discussed in §3.5). Thus, JENNER can determine the answers returned in both cases when the probability of $t_i.\mathcal{A}_j$ satisfying the query condition is p_{low} or it is p_{high}. Let these answers be Ans_{low} and Ans_{high} respectively.

We can now determine the estimated quality of the answer after execution of f_k on $t_i.\mathcal{A}_j$ as a weighted sum of the quality of the potential answers $\widehat{Qty}(Ans_{low})$ and $\widehat{Qty}(Ans_{high})$).

$$p_{w-1}\widehat{Qty}(Ans_{high}) + (1 - p_{w-1})\widehat{Qty}(Ans_{low}) \tag{9}$$

where p_{w-1} refers to the probability of $t_i.\mathcal{A}_j$ satisfying the query condition in its state in D_{w-1}.

Given the above expected quality of answers after execution of f_k on $t_i.\mathcal{A}_j$, we can now determine the benefit of its execution to the results. Note that such a value could potentially be negative depending upon the value of p_{w-1}. In such a case, we consider its benefit to be 0 and such a function would not be chosen for enrichment.

3.2.2 *Generalizing to Other Queries* .

To estimate benefit for general queries, we have to extend the model for both estimating the quality of query result in the previous epoch e_{w-1} and also the benefit of executing the triples in enrichment plan EP_w of the current epoch. Let us consider a query Q with conditions on n relations $R_1, R_2, \ldots, R_n$. For each R_i, there could be multiple selection and join conditions on both fixed and derived attributes.

At epoch e_{w-1}, the tuples of R_i are classified as one of the following two types: (*i*) the tuples that have met the selection condition on derived attributes of R_i, denoted by R_i^σ, [4] and (*ii*) the tuples that do not satisfy such selection conditions, are denoted by $R_i^{\neg\sigma}$. The tuples of R_i^σ are further classified as those that were part of the answer set (*i.e.*, one of the tuple in the answer set was generated by this tuple) or not in the answer set (if there does not exist any tuple in the query's answer so far from this tuple).

In order to determine R_i^σ, we determine the set of tuples in Ans_{w-1} and look at the tuples of R_i with the minimum *match_prob*, *i.e.*, probability of satisfying all the selection conditions of R_i, which is part of Ans_{w-1}. We refer to this minimum *match_prob* as the *relation-threshold* of R_i. All the tuples with *match_prob* higher than this threshold compose R_i^σ and all the tuples with *match_prob* lower

[4]If there are no selection conditions on derived attributes then all the tuples are part of R_i^σ.

than the threshold compose $R_i^{\neg\sigma}$. The candidate tuples are chosen from $R_i^{\neg\sigma}$ of the relations.

To compute the probability of a tuple $t_i \in Ans_w$ to be part of the answer Ans^{real}, let us consider the projection of t_i to its constituent tuples in the relations of $R_1, R_2, \ldots, R_n$. Let us denote the corresponding tuple in R_j as $t_i[R_j]$. The probability of $t_i[R_j]$ satisfying the selection condition in Q on attributes in R_j is computed as discussed above in the context of selection queries. The probability of the join condition between two relations R_j and R_k (say $R_j.\mathcal{A}_m = R_k.\mathcal{A}_m$) being satisfied by t_i can be computed as follows: Let the determinized value of $t_i[R_j]$ be $Det(t_i[R_j])$. Its probability of $t_i[R_j]$ satisfying the join condition on the derived attribute $\mathcal{A}_m$ is $Val(state(t_i[R_j].\mathcal{A}_m)[Det(t_i[R_j].\mathcal{A}_m)]$. Likewise, suppose the determinized value of $t_i[R_k]$ be $Det(t_i[R_k])$. The probability of $t_i[R_k]$ satisfying the join condition on the attribute $\mathcal{A}_m$ is $Val(state(t_i[R_k].\mathcal{A}_m)[Det(t_i[R_k].\mathcal{A}_m)]$. Hence, the probability of tuple t_i to satisfy the join condition of $R_j.\mathcal{A}_m = R_k.\mathcal{A}_m$ is the product of the above two probabilities, *i.e.*, $Val(state(t_i[R_j].\mathcal{A}_m)[Det(t_i[R_j].\mathcal{A}_m)] \cdot Val(state(t_i[R_k].\mathcal{A}_m)[Det(t_i[R_k].\mathcal{A}_m)]$. The overall probability of tuple t_i is computed by computing the product of all the probabilities for the predicates present in Q.

Given the probability of all the tuple t_i in Ans_w to be part of the real answer set Ans^{real}, we compute F_α measure using Equation 5.

In order to compute the benefit of triple $\langle t_i, \mathcal{A}_j, f_k \rangle$, where tuple t_i belongs to relation R_p and it is part of $R_p^{\neg\sigma}$, we need to estimate the quality of the answer that would result if Q were to be executed on the database after executing f_k on $t_i.\mathcal{A}_j$. We follow the same strategy, as in selection query, we generate p_{low} and p_{high} for the condition on $t_i.\mathcal{A}_j$ to be met. For each of the case, we re-execute the query, generate the answers Ans_{high} and Ans_{low}. As in the case of selection queries, we execute the pruning of the answers to maximize F_α measure §3.5.

The above way of computing benefit for executing f_k on $t_i.\mathcal{A}_j$ requires first for us to determine the probability values p_{low} and p_{high} from the entropy reduction of f_k, re-execute the query Q on the database state resulting $D_{e_{w-1}}$ with the probability of $t_i.\mathcal{A}_j$ matching the query condition appropriately modified to p_{low} (p_{high}), and then finally to run the produce query result in both cases (the complexity, as will become clear in §3.5 is $|Ans|log(|Ans|)$, where $|Ans|$ is the size of the answer from which is the query result is selected. Thus, the complexity of the above can be estimated as $O(costQ + |Ans_w|log(|Ans_w|))$, where $costQ$ is the time taken to execute the query, $|Ans_w|$ is the size of the answers returned. As a result, the overall complexity of benefit estimation is $O(n(costQ + |Ans_w|log(|Ans_w|))$ where n is the size of $CandidateSet^M$.

3.3 Selecting Enrichment Plan

This step chooses a set of ⟨tuple, derived attribute, enrichment function⟩ triples as the enrichment plan of epoch e_w. The problem of selecting an enrichment plan is a budgeted Knapsack problem as we need to find an enrichment plan with total cost less than or equal to *epoch_duration* and that has maximum sum of benefit values among all the possible subset of ranked tuples. The algorithm needs to choose a set of ⟨tuple, enrichment function⟩ with highest summation of benefit values and a total cost not exceeding the duration of epoch. We use a greedy approach to choose this enrichment plan for the epoch e_w. For the ⟨tuple, derived attribute, enrichment function⟩ triples in $CandidateSet^M$, we compute their benefit as described in the previous step of plan generation phase and sort them based on their benefit values.

The approach needs to find an enrichment plan with total cost less than or equal to *epoch_duration* with the maximum sum of benefit values among all possible subset of ranked tuples. This is a budgeted Knapsack problem for which we use a greedy approach to solve it. The triples in CandidateSet are sorted in decreasing order of their benefit values based on the benefit metric. The enrichment plan is chosen from the sorted set starting from the triple with highest benefit. Once the running total cost of the chosen triples exceeds *epoch_duration*, the WSPT algorithm terminates and the chosen set is considered as the enrichment plan.

3.4 Execution of Enrichment Plan

In this step, the ⟨tuple, derived attribute, enrichment function⟩ triples present in EP_w are executed. For each tuple $t_i \in EP_w$, JENNER updates the state of t_i. Next, the determinized representation of t_i is updated based on the output of all the enrichment functions executed on it. For each tuple for which a derived attribute value was enriched, the *NBF* function for that attribute changes. Hence, JENNER, calculates the new benefit of the tuple, if it is enriched using the *NBF* function at the current state. These updated benefit of the tuples that were enriched in epoch e_w are reflected in the $CandidateSet^M$ data structure. Hence, the next epoch can choose an enrichment plan by comparing the updated benefit of enriched tuples in e_w and the previous benefit of tuples that were not enriched.

3.5 Produce Query Result

After updating the state of tuples enriched in the current epoch e_w, the original query Q is re-executed on the determinized representation to determine the set of potential answer. For each tuple t_i in Ans_w computed, we determine the probability of t_i to be in Ans^{real}, based on the probability of tuples in the base relation R_i used to construct t_i. Instead of returning all tuples in Ans_w, we return a subset of them, in order to maximize the answer quality based on the quality metric (*i.e.*, F_α-measure for set-based queries). [5]

JENNER is based on the following observation (proved in a theorem in Appendix 2) : Let $t_1 t_2, \ldots, t_n$ be the set of tuples in Ans_w sorted based on their probability of being in Ans^{real}. The $E(F_\alpha)$ measure of the query result increases monotonically with the inclusion of more tuples t_i starting with the highest probability value up to the inclusion of a certain tuple; beyond which the $E(F_\alpha)$ measure decreases monotonically with the inclusion of any more tuples.

We utilize this observation, we sort results based on probability and continue including answer tuples until $E(F_\alpha)$ measure of the answer keeps increasing. We stop when $E(F_\alpha)$ drops. We refer to the probability of the last tuple that is part of the answer set in epoch e_w as the *answer-threshold* probability. The complexity of this step has $O(nlog(n))$ where n is the size of $CandidateSet^M$.

[5]For aggregation queries, we first determine the set of answers that optimizes F_α-measure and then compute the aggregation function.

Example 3.3. Let us consider the query of Figure 5a containing conditions on two relations of R_1 and R_2. Suppose R_1 contains five tuples that are in the probe query result of R_1: $\langle r_1^1, r_2^1, \ldots, r_5^1 \rangle$ and relation R_2 contains ten tuples in its probe query results as follows: $\langle r_1^2, r_2^2, \ldots, r_{10}^2 \rangle$. Without loss of generality, let us consider that the tuples of reach relation are sorted based on their probability of satisfying all the selection conditions on the derived attributes. Considering the possible tuple pairs of $\langle\langle r_1^1, r_1^2 \rangle, \ldots, \langle r_5^1, r_{10}^2 \rangle\rangle$, JENNER computes the probability of the tuples as described in Example 3.4. It keeps including the tuples in Ans_w as long as the $E(F_1)$ measure of the answer set keeps increasing. The tuple beyond which the $E(F_1)$ measure decreases, are not included in the answer set. The Ans_w chosen in this way has maximum $E(F_1)$ measure as described above.

We next discuss how the probability of a tuple $t_i \in Ans_w$ to be in the true answer based on the corresponding tuples in the base relations $R_i \in Q$ that resulted in the answer tuple t. For simple selection queries over derived attributes, it is simply the probability of the value of the tuple to match the query condition (*i.e.*, $Val(state(t.\mathcal{A}_j)) = a_j$ for a selection condition of $\mathcal{A}_j = a_j$).

For multiple selection conditions on derived attributes of a relation, the probability is estimated under the assumption derived attributes to be independent. For queries where answer tuple t_i is formed using tuples from multiple relations, the corresponding probability is based on the product of the probabilities of individual tuples to satisfy the individual selection and the join conditions. [6] An example of computing the probability of the tuples satisfying the query Q is shown below.

Example 3.4. Let us consider the query of Figure 5a on relations R_1 and R_2 and two tuples $r_1 \in R_1$ and $r_2 \in R_2$ which were part of the probe query results of R_1 and R_2. Suppose, the value of $r_1.\mathcal{A}_1$ is a_1 (*i.e.*, the value with highest probability after determinization) and the probability associated with the value is 0.9. Similarly, let the value of $r_2.\mathcal{A}_6$ is a_6 and the probability associated with it is 0.8. Now considering the join condition on derived attribute (*i.e.*, $R_1.\mathcal{A}_3 = R_2.\mathcal{A}_3$), suppose the attribute values of them after determinization match and the corresponding probability values are 0.95 and 0.85. Hence, the probability of the tuple pair $\langle r_1, r_2 \rangle$ satisfying all the query conditions in Q are as follows: $\mathcal{P}(\langle r_1, r_2 \rangle) = 0.9 \times 0.8 \times 0.95 \times 0.85 = 0.58$. We compute these probability values for all the tuple pairs in the probe query result of R_1 and R_2 and whose determinized representation of the derived attribute matches with the conditions in the query.

After determining the Ans_w that optimizes the $E(F_\alpha)$ metric, we prune tuples from the candidate set whose impact on improving the quality of the answer set in subsequent epochs is expected to be low. This is achieved by finding out the tuples of each relation that already contributed to Ans_w of the current epoch and remove them from $CandidateSet^M$. Enriching such tuples have low impact on improving the quality of the query result as proved in a theorem in Appendix 6.3.

[6] In case of duplicates, the probability values are added up as in probabilistic databases [12]

3.6 Optimizing Benefit Computation

The techniques for benefit computation for an enrichment function f_k on attribute $t_i.\mathcal{A}_j$ described in §3.2.1 and §3.2.2 used simulated execution of f_k to assess the impact of its execution on the overall quality of the answers resulting the overall complexity of $O(n(costQ + |Ans_w| log(|Ans_w|))$ where n is the size of $CandidateSet^M$.

We now present a strategy wherein *SelectEnrichmentPlan* can select the enrichment plan without explicitly calculating the benefit of the enrichment functions. We do so first for selection queries and then generalize to other queries.

Selection Query. In the modified strategy, for each $\langle t_i, \mathcal{A}_j, f_k \rangle \in CanddiateSet^M$, we compute the *RelativeBenefit* defined below.

$$RelativeBenefit(t_i, \mathcal{A}_j, f_k) = \frac{\mathcal{P}_i(\mathcal{P}_i + \Delta\mathcal{P}_i)}{c_k} \quad (10)$$

where $\mathcal{P}_i$ is the probability of the tuple satisfying the selection conditions and $\mathcal{P}_i + \Delta\mathcal{P}_i$ is the new probability of tuple t_i if it is enriched in the current epoch. Here, Δ would be positive for p_{high} and negative for p_{low}.

We can show that if a triple has higher relative benefit than another, then the first triple will always have a higher benefit (viz., Equation 7) as stated in the following theorem.

Theorem 1. *A triple (t_i, $\mathcal{A}_j$, f_k) has higher benefit than a triple (t_q, $\mathcal{A}_s$, f_v) in epoch w irrespective of the values of $\mathcal{P}_i$, $\mathcal{P}_q$, $\Delta\mathcal{P}_i$ and $\Delta\mathcal{P}_q$ if the following condition holds:*

$$RelativeBenefit(t_i, \mathcal{A}_j, f_k) > RelativeBenefit(t_q, \mathcal{A}_s, f_v) \quad (11)$$

Proof. We prove this theorem as follows: for the given values of $\mathcal{P}_i$, $\mathcal{P}_q$, $\Delta\mathcal{P}_i$ and $\Delta\mathcal{P}_q$, there can be four possible orders among them. They are as follows: (*i*) $\mathcal{P}_i > \mathcal{P}_q$ and $\Delta\mathcal{P}_i > \Delta\mathcal{P}_q$, (*ii*) $\mathcal{P}_i > \mathcal{P}_q$ and $\Delta\mathcal{P}_i < \Delta\mathcal{P}_q$, (*iii*) $\mathcal{P}_i < \mathcal{P}_q$ and $\Delta\mathcal{P}_i > \Delta\mathcal{P}_q$, (*iv*) $\mathcal{P}_i < \mathcal{P}_q$ and $\Delta\mathcal{P}_i < \Delta\mathcal{P}_q$. For each of these orders, we determine the answer set and calculate the quality of the answer when the triple (t_i, $\mathcal{A}_j$, f_k) is executed. Similarly, we calculate the quality when triple (t_q, $\mathcal{A}_s$, f_v) is executed. We measure their benefit values using Equation 7.

Let m_1 be the number of tuples of Ans_{w-1} that move out of Ans_w as a result of increased probability of tuple t_i satisfying the query from $\mathcal{P}_k$ to $\hat{\mathcal{P}}_k$, where $m_1 \geq 0$. Furthermore, let $m_2 \geq 0$ be the number of tuples of Ans_{w-1} that move out of Ans_w as a result of changing the probability of t_q from $\mathcal{P}_q$ to $\hat{\mathcal{P}}_q$.

Given three possible cases of m_1 and m_2 (*i.e.*, $m_1 > m_2$, $m_1 < m_2$, and $m_1 = m_2$), we consider the possible combinations (16 possible combinations) of the values of $\mathcal{P}_k$, $\mathcal{P}_q$, $\Delta\mathcal{P}_k$, and $\Delta\mathcal{P}_q$ and show if Equation 11 holds, then the benefit of $(t_i, \mathcal{A}_j, f_k)$ will be higher than the benefit of $(t_q, \mathcal{A}_s, f_v)$. In the following we provide the proof of these scenarios:

For ease of notation, we take the following steps: Since, $E(F_\alpha(Ans_{w-1})) = \frac{(1+\alpha)(\mathcal{P}_1+\cdots+\mathcal{P}_\tau)}{\alpha(\mathcal{P}_1+\mathcal{P}_2+\cdots+\mathcal{P}_{|O|})+\tau}$, this expression is simplified as $\frac{X}{Y+\tau}$. We denote the value of $\frac{\mathcal{P}_k}{c_k}$ by v_k and the value of $\frac{\mathcal{P}_q}{c_v}$ by v_q.

Simplifying the expression of benefit the triples (*i.e.*, Equation 7) and quality measured using $\hat{F}_\alpha$ measure (*i.e.*, Equation 5), benefit of triple $\langle t_i, \mathcal{A}_j, f_k \rangle$ is higher than the triple $\langle t_q, \mathcal{A}_s, f_v \rangle$, when the following condition holds:

$$v_k(\frac{X - (1+\alpha)\cdot(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)}) + (1+\alpha)\cdot(\mathcal{P}_k + \Delta\mathcal{P}_k)}{Y + (\tau - m_1) + \alpha\cdot\Delta\mathcal{P}_k}) > v_q(\frac{X - (1+\alpha)\cdot(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)}) + (1+\alpha)\cdot(\mathcal{P}_q + \Delta\mathcal{P}_q)}{Y + (\tau - m_2) + \alpha\cdot\Delta\mathcal{P}_q}) \quad (12)$$

Case 1: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1 > m_2$. Comparing the denominators of Equation 12, we observe that $(\tau - m_1) < (\tau - m_2)$ and $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$. This implies that the denominator on the L.H.S. is smaller than the denominator on the R.H.S. In the numerator of L.H.S., the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ is higher than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is higher than m_2. Furthermore, if $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ then the numerator of the L.H.S. will be higher than the numerator of the R.H.S. Thus we conclude that Equation 12 is satisfied when condition $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied.

Case 2: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1 > m_2$. In Equation 12, the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ on the L.H.S is higher than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is higher than m_2. In the denominator, although the value of $\Delta\mathcal{P}_k$ is higher than $\Delta\mathcal{P}_q$, the total value of $(\tau - m_1 + \alpha\cdot\Delta\mathcal{P}_k)$ is lower than $(\tau - m_2 + \alpha\cdot\Delta\mathcal{P}_q)$ as both $\Delta\mathcal{P}_k$ and $\Delta\mathcal{P}_q$ are less than one.

Case 3: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1, m_2 = 0$. Let us compare the L.H.S and R.H.S. of Equation 12. In the numerator, if the term of $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k)$ is higher than the value of $v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$, then the numerator of L.H.S will be higher than the numerator of R.H.S. Hence, the value of the expression in the left hand side will be higher.

Case 4: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1, m_2 = 0$. In Equation 12, after simplifying some steps further, we derive that the condition in which the L.H.S. will be higher than the R.H.S. is as follows: $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k)\Delta\mathcal{P}_q > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)\Delta\mathcal{P}_k$. According to the assumption $\Delta\mathcal{P}_q$ value is higher than the value of $\Delta\mathcal{P}_k$. This implies that, if the condition $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied, then L.H.S. will be higher than the R.H.S.

The above proofs will also hold for the scenarios where $m_1 = m_2$ and $m_1 > 0$. Only difference will be as follows: an additional constant term (*i.e.*, $\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + \cdots + \mathcal{P}_{\tau-(m_1-1)}$) will be added to the numerators of both the sides of Equation 12. The remaining steps will remain the same as the proofs of Cases 1-4.

Case 5: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1 < m_2$. Comparing both the denominators of Equation 12, we can see that $(\tau - m_1) < (\tau - m_2)$ and $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$. This implies that the denominator of L.H.S. is lower than the denominator of R.H.S. In the numerators, the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ is lower than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is less than m_2. This condition makes the numerator of L.H.S higher than R.H.S. Furthermore, if $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied then the numerator of L.H.S. is higher than R.H.S.

Case 6: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1 < m_2$. From Equation 12, we can derive the following equation:

$$v_k(X - (1+\alpha)\cdot(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)}) + (1+\alpha)\cdot(\mathcal{P}_k + \Delta\mathcal{P}_k))\cdot(Y + (\tau - m_2) + \alpha\cdot\Delta\mathcal{P}_q) > v_j(X - (1+\alpha)\cdot(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)}) + (1+\alpha)\cdot(\mathcal{P}_q + \Delta\mathcal{P}_q))\cdot(Y + (\tau - m_1) + \alpha\cdot\Delta\mathcal{P}_k) \quad (13)$$

In the above equation, the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ is lower than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is smaller than m_2. This favors the value in the left hand side of the equation. Furthermore, if the value of $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k)$ is higher than the value of $v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$, then the benefit of first triple is higher than the second triple.

Case 7: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1 < m_2$. Comparing the L.H.S. of Equation 13 with the R.H.S., we observe that if the value of $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k)$ is higher than the value of $(\mathcal{P}_q + \Delta\mathcal{P}_q)$, then the whole expression of L.H.S. becomes higher than the R.H.S.

Case 8: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q, \mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$, and $m_1 < m_2$. Let us consider Equation 12 and compare the L.H.S. with the R.H.S. After simplifying them, we can derive that the condition in which the left hand side will be higher than the right hand side is as follows: $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k)\Delta\mathcal{P}_q > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)\Delta\mathcal{P}_k$. According to the assumption of this case, $\Delta\mathcal{P}_q$ value is higher than the value of $\Delta\mathcal{P}_k$. This implies that, if the condition $v_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > v_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ holds, then Equation 12 is satisfied and the benefit of first triple is higher than the second triple.

The above proofs (*i.e.*, the proofs of Cases 5-8) will also hold for the scenarios where $m_1 > m_2$, due to the symmetric nature of the assumptions. Based on the proofs of Cases 1-8, we conclude that given two triples $(t_i, \mathcal{A}_j, f_k)$ and $(t_q, \mathcal{A}_s, f_v)$, if *RelativeBenefit*$(t_i, \mathcal{A}_j, f_k)$ > *RelativeBenefit*$(t_q, \mathcal{A}_s, f_v)$ then the first triple has higher benefit than the second triple.

Given the above theorem, JENNER computes the *RelativeBenefit* of the triples and select enrichment plan (§3.3) based on relative benefit. Note that, the complexity of computing the *RelativeBenefit* for all the triples in the candidate set is $O(n)$, where n is the size of the *CandidateSet*^{M} which is significantly lower compared to the approach based on computing benefits explicitly.

More General Queries. To exploit a strategy based on relative benefit for more general queries, we need to estimate the number of tuples that would be resulting from a tuple t_i in relation R_p. JENNER estimates the average number of tuples that were generated by the tuples in R_p^σ in the answer of Ans_{w-1}. We refer to this value as $\lambda^{w-1}(R_p)$. We measure the relative benefit of $\langle t_i, \mathcal{A}_j, f_k \rangle$, where $t_i \in R_p$ as follows:

$$\mathit{RelativeBenefit}(t_i, \mathcal{A}_j, f_k) = \lambda_{R_i}\cdot[\frac{\mathcal{P}_i(\mathcal{P}_i + \Delta\mathcal{P}_i)}{c_k}] \quad (14)$$

The relative benefit reflects the amount of improvement in the quality of the query result that is achieved by the answer tuples generated from tuple t_i in R_p. As for selection queries, the enrichment plan EP_w is chosen using the above *RelativeBenefit* metric.

4 EXPERIMENTAL EVALUATION

Our experimental study evaluates JENNER along the following aspects.

ID	Query
Q1	SELECT * from MultiPie where gender=1 and CameraID <12
Q2	SELECT * from MultiPie where gender=1 and expression = 2 and CameraID <12
Q3	SELECT tid, UserID, Tweet, location, TweetTime from TweetData where sentiment = 1 and and topic = 2 and TweetTime between('16:00','18:00')
Q4	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
Q5	SELECT * from TweetData T1, TweetData T2 where T1.sentiment = T2.sentiment and T1.TweetTime between('16:00','18:00') and T2.TweetTime between ('16:00','18:00')
Q6	SELECT * from TweetData T1, State S where T1.location = S.city and S.state='California' and T1.sentiment = 1 and T1.TweetTime between('16:00','18:00')
Q7	SELECT gender, count(*) from MultiPie where CameraID <12 group by gender
Q8	SELECT topic, count(*) from TweetData where TweetTime between('16:00','18:00') group by topic

Table 3: Queries used.

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[41]	100K	16.9	gender(2)	DT,GNB,KNN,MLP
			expression(5)	DT, GNB, RF, KNN

Table 4: Datasets used in experiments.

- quality improvement due to JENNER 's benefit based approach compared to sampling-based approaches (discussed later) for selecting enrichment.
- Overhead of enrichment plan generation both in terms of time and storage cost.
- Impact of different epoch sizes on the quality of query results.
- Impact of pruning of *CandidateSet* and calculating benefit using *RelativeBenefit* metric.

Datasets. We used the following datasets to evaluate JENNER: (*i*) TweetData containing 11 million rows with two derived attributes: ⟨`sentiment and topic`⟩, and six fixed attributes: ⟨`tid`, `UserID`, `Tweet`, `feature`, `location`, and `TweetTime`⟩ (*ii*) MultiPie [41] dataset with 500K facial images, two derived attributes: ⟨`gender and expression`⟩, and five fixed attributes: ⟨`ImageID`, `CameraID`, `CameraLocationID`, `Image`, `ImageTime`⟩ (see Table 4).

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); as enrichment functions. The GNB classifier was calibrated using isotonic-regression model [47] and all the other classifiers were calibrated using Platt's sigmoid model [37] during cross-validation. After calibration, each classifiers outputs a real probability distribution based on the validation data. The classifiers trained for the attributes are shown in Table 4.

Queries. As shown in Table 3, we selected eight queries, where ***Q1-Q3*** are ***selection queries***, ***Q4-Q6*** are ***join queries***, and ***Q7-Q8*** are ***aggregation queries*** with number of groups as 2 (low) and 40 (high) respectively.

Q	JENNER	FO	OO	RO	Q	JENNER	FO	OO	RO
Q1	**0.87**	0.36	0.33	0.32	Q5	**0.73**	0.39	0.35	0.33
Q2	**0.84**	0.34	0.32	0.31	Q6	**0.72**	0.37	0.36	0.32
Q3	**0.76**	0.43	0.35	0.31	Q7	**0.86**	0.35	0.32	0.31
Q4	**0.80**	0.34	0.33	0.31	Q8	**0.74**	0.37	0.33	0.34

Table 5: Progressive Scores.

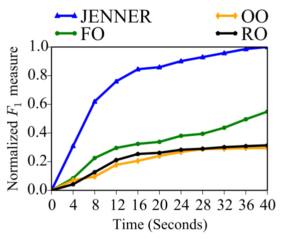
Implementation Details. In the implementation, we use two servers: (*i*) an enrichment server and (*ii*) a DBMS server. Given a query Q, the enrichment server generates a set of probe queries and executes them on the DBMS to fetch the tuples that may require enrichment. The state of the tuples are replicated in both the DBMS and the enrichment server. This reduces communication without adding much storage overhead since the size of state tables are usually much lower than the size of data tables (as shown later in experiments). The enrichment is performed at the enrichment server in epochs resulting in changes to following tables: (*i*) the state tables stored in the enrichment server as well as the one stored in the database, and (*ii*) the combined derived attribute values in the data tables. At the end of an epoch, JENNER returns a query results to the user based on the updated derived attribute values.

Enrichment Plan Generation Strategies. For experiments, we compare JENNER with three different sampling based plan generation strategies: (*i*) *Sample-based with Object Order* (**OO**): that randomly selects tuples from the set of tuples satisfying predicates on fixed attributes. Selected tuples are completely enriched by executing all enrichment functions available for derived attributes present in the query. (*ii*) *Sample-based with Function Order* (**FO**): that selects enrichment functions based on the decreasing order of their $\frac{quality}{cost}$ values. The top-most function from the sorted enrichment functions is executed on all tuples (obtained after checking the predicates on fixed attributes), before executing the next function. (*iii*) *Sample-based with Random Order* (**RO**): that selects both tuples and enrichment functions randomly from a set of ⟨tuple, function⟩ pairs after checking predicates on fixed attributes.

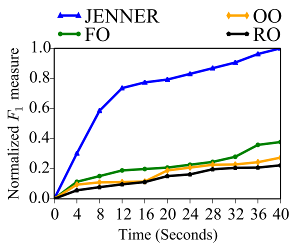
4.1 Experimental Results

Experiment 1 (Progressiveness Achieved). This experiment compares JENNER with FO, OO, and RO approaches. We set the epoch size for Q1, Q2, Q4 to 4 seconds and for the remaining queries, to 8 seconds. Progressive score is computed as a weighted summation of F_1 measures with weight of the epoch e_w set as $(1 - \frac{w}{w_{max}})$ where the w_{max} (corresponding to the maximum number of epochs) is set as 15. The results are shown in Figure 3 where we measure the quality of the query result using *normalized F_1 measure* for set based queries Q1-Q6 and *normalized root-mean-square-error* (RMSE) for aggregation queries Q7-Q8. The normalized F_1 measure is calculated as F_1/F_1^{max}, where F_1^{max}[7] is the maximum F_1 measure that is achievable by executing all the enrichment functions. Similarly, normalized root-mean-square-error is calculated by measuring $RMSE/RMSE^{min}$ where $RMSE^{min}$ is the minimum RMSE

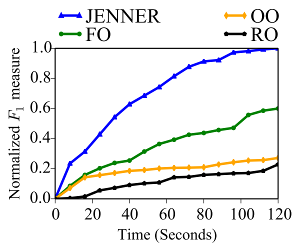
[7]Recall that our evaluation of F_1 measure in experiments is based on the ground truth data since we have access to them.



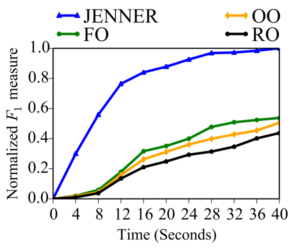
(a) Multi-PIE (Q1).



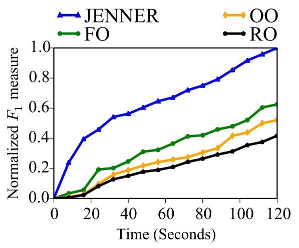
(b) Multi-PIE (Q2).



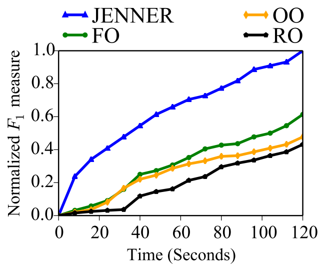
(c) TweetData (Q3).



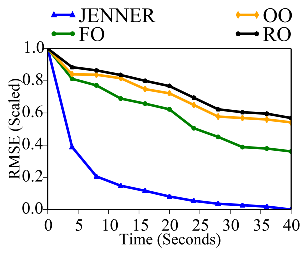
(d) Multi-PIE (Q4).



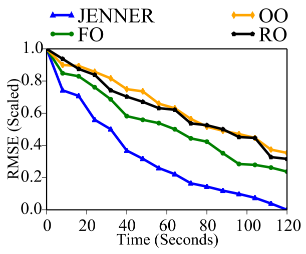
(e) TweetData (Q5).



(f) TweetData (Q6).



(g) MultiPie (Q7).



(h) TweetData (Q8).

Figure 3: Performance results of different plan generation strategies.

achievable by executing all the enrichment functions. Furthermore, we report the progressive scores in Table 5.

Figure 3 and Table 5 show that JENNER outperforms the sampling based approaches significantly for all the queries. With JENNER , the answer achieves a high quality within the first few epochs of the execution as shown in Figure 3 for all queries. For example, Figure 3(a) shows that JENNER achieves nF_1-measure 0.9 within the first 20 seconds of query execution. Furthermore, JENNER achieves a high rate of quality improvement in the epochs in the beginning resulting in the highest progressive score compared to the sampling based approaches. We observe this performance gap as JENNER monitors and reassesses the progress of enrichment at the beginning of each epoch to identify and execute the triples that are expected to yield the highest improvement in quality. In Table 5, we observed that JENNER achieves high progressive scores for all the queries. For a given query, another alternative is to enrich the tuples completely. The progressive scores are as follows: 0.12, 0.14.

Experiment 2 (Overhead). Table 6 measures the *time* and *storage* overheads of enrichment plan generation in JENNER . The former is measured by calculating the average time taken by the enrichment plan generation phase for different queries. The results show that (*i*) as expected, plan generation overhead increases as more tuples satisfy the query predicates (*e.g.*, see increase in planning time for $Q2$ and $Q3$ that have 500 vs. 10,000 tuples in $CandidateSet^M$) ; (*ii*) Irrespective of the selectivity, the plan generation time remains a small fraction of the overall execution time of queries (ranging from 0.3 to 1.55%). Thus, JENNER 's adaptive approach of selecting tuples to enrich does not impose significant overhead. In terms of storage overhead, the $CandidateSet$, $CandidateSet^M$, and EP_w maintained by JENNER were less than 10 MB which is a small fraction of the data sizes (which were 10.5 an 16.9 GB as shown in Table 4). Furthermore, the state table sizes for the tuples in `TweetData` and MultiPie data were 0.9 GB and 0.1 GB, respectively.

Experiment 3 (Epoch Size). The objective of this experiment is study how choice of epoch size effects progressiveness achieved

Query	Plan Gen. Time	%age of Total Time	Query	Plan Gen. Time	%age of Total Time
Q1	0.12	0.3	Q5	1.58	1.32
Q2	0.26	0.65	Q6	0.85	0.71
Q3	1.4	1.17	Q7	0.62	1.55
Q4	0.58	1.45	Q8	1.6	1.33

Table 6: Average plan generation time (in seconds).

Query	JENNER	Naive	Query	JENNER	Naive
Q1	500	1000	Q5	11000	20000
Q2	1000	2000	Q6	6000	10000
Q3	8000	20000	Q7	500	1000
Q4	1200	2000	Q8	7000	10000

Table 7: Average Number of Candidates in $CandidateSet^M$.

by JENNER for different queries. Figure 4 plots: 1) Time to reach (TTR) 90% quality by JENNER for Q2 and 2) Overhead as percentage of total query execution time. From Figure 4(a), we observe that when we reduce epoch size for Q2 from 8 to 4 seconds, the TTR value reduces. With reducing epoch size, JENNER becomes more adaptive choosing enrichment plans more frequently. This results in improved choices and hence increased progressiveness. However, when we reduce epoch size even further from 4 to 2 seconds, the overheads due to frequent plan generation (which changes from 0.65 percent to 6 percent) reduces the effective time available to enrich data, thereby reducing the progressiveness achieved, thus increasing the TTR. While we focused the experiment description on query Q2, the behavior of other queries is similar though the optimal point (which for Q2 is 4 seconds) would differ from query to query. Given the impact of choosing epoch size to query performance, exploring a strategy that chooses epoch size based on query characteristics (as compared to a fixed strategy used in current JENNER) would be an interesting direction of exploration.

Experiment 4 (Impact of Pruning). As mentioned in §3.2, JENNER restricts the choice of tuples to enrich in the enrichment plan to

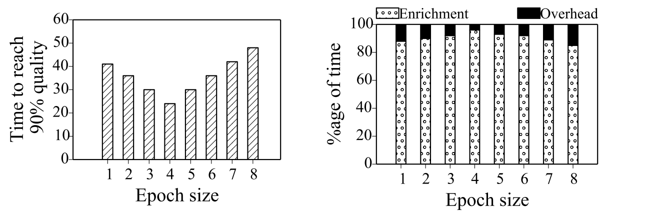


Figure 4: Effect of epoch size: (a) time to reach 90% of quality for Q2 and (b) percentage of time spent in enrichment plan generation (overhead) as compared to enrichment.

Query	rel. benefit	benefit	Query	rel. benefit	benefit
Q1	0.3%	25.45%	Q5	1.32 %	94.17 %
Q2	0.65 %	87.75 %	Q6	0.85 %	58.96 %
Q3	1.17 %	69.38 %	Q7	0.62 %	43.14 %
Q4	1.45%	100%	Q8	1.6 %	98.34%

Table 8: Comparison of Percentage of time Taken by Enrichment Plan Generation of the Total Execution Time for Different Strategies in JENNER .

only those that are not in the answer set of the previous epoch. We compare this strategy with the strategy of considering all tuples in $CandidateSet(R_i)$ for generating enrichment plans. Table 7 shows that the size of $CandidateSet^M$ in JENNER is significantly smaller therefore reducing the cost of enrichment plan generation. The entries in $CandidateSet^M$ that were pruned by JENNER were (almost) never chosen for enrichment. As a result, this cost reduction comes with no impact on the quality achieved by JENNER

Experiment 5 (Impact of Optimized Benefit Estimation). This experiment compares JENNER when it uses a the naive strategy (described in §3.2) for benefit estimation that has higher complexity with JENNER using the estimation approach optimized for benefit described in §3.6 (with complexity $O(n)$). Table 8 plots the percentage of time taken by enrichment plan generation of the total execution time when the benefit estimation is done using the naive versus optimized strategy. The figure shows that the naive strategy would have required 25% to 100 % of the total execution time just for benefit estimation, thereby making JENNER impractical. The optimized strategy for benefit estimation requires only a small fraction of the total execution time, enabling JENNER to allocate most of the time to enrich data and process query. As shown in Experiment 1, it results in significantly improved progressive scores.

5 RELATED WORKS

Our approach in JENNER related to several lines of prior research. Progressive query answering has been explored in approximate processing of aggregate queries [15]. Such techniques offer error bounds based on sampling [3, 36] which improves as larger samples are considered. Such techniques, however, do not consider the problem of enrichment in query processing and cannot be used in our setting. Progressive data processing has also been considered in data cleaning contexts such as entity resolution [6, 29, 35]. JENNER adapted the metric for progressive enrichment from these works on progressive data cleaning. JENNER (which explores progressive enrichment during query processing) differs from the above listed research which has not considered progressive data cleaning in the context of queries. As mentioned in §1, data cleaning during query processing has been studied in [39]. However, unlike JENNER these works have not considered progressive data processing.

JENNER is also related to work on expensive predicate optimization in [20, 24, 27]. These works have focused on both static and dynamic predicate reordering during query optimization / processing so as to minimize cost of query processing. In contrast, JENNER focused on progressive enrichment during query processing.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we describe an approach, entitled JENNER, that optimizes data enrichment with progressive query processing. JENNER overcomes several limitations of both offline and at-ingest enrichment by optimally integrating enrichment during query processing. To overcome the increased query latency, JENNER exploits tradeoff between quality and efficiency that is implicit in the realization of enrichment functions that are typically based on machine learning/signal processing techniques. Furthermore, JENNER hides latency by supporting progressive query answering that refines as data gets enriched. Our experimental section validates improvement achieved by JENNER over naive strategies to support progressiveness in query processing.

JENNER opens several new directions of research worthy of future exploration. In JENNER during each epoch queries are reexecuted even though the modified data is limited to only that which is enriched during the epoch. While JENNER is designed for situations where dominant cost is enrichment (and not query processing), nonetheless, we could improve execution by exploiting incremental query processing methods as in [1, 22, 32]. Such an extension will be of significant interest specially in the context where query processing costs and enrichment costs are comparable. As another extension, JENNER has been developed when determinization functions used on uncertain data return a single value. Prior work such as [26] have explored advantages of considering determinization functions that may allow multiple values (interpreted using OR semantics) to be returned and to let the query processor choose the final representation. Such an approach can improve quality of answer further. Our initial attempt to incorporating such enhanced enrichment functions are included in the longer version of this paper. Supporting clustering based functions in JENNER is also an interesting future work. Finally, mechanisms to embed JENNER into existing database systems to support interactive enrichment is an interesting direction of future exploration.

REFERENCES

[1] Incremental view maintenance development for postgresql. https://github.com/sraoss/pgsql-ivm.
[2] Internet live stats. http://www.internetlivestats.com.
[3] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. EuroSys '13.
[4] S. Agrawal et al. Scalable ad-hoc entity extraction from text collections. *Proc. VLDB Endow.*, 1(1):945–957, Aug. 2008.
[5] A. Alsaudi, Y. Altowim, S. Mehrotra, and Y. Yu. TQEL: framework for query-driven linking of top-k entities in social media blogs. *Proc. VLDB Endow.*, 14(11):2642–2654, 2021.
[6] Y. Altowim et al. Progressive approach to relational entity resolution. *VLDB '14*.
[7] Y. Altowim et al. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 2014.
[8] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *Proc. VLDB Endow.*, 2015.

[9] L. Becker et al. Avaya: Sentiment analysis on twitter with self-training and polarity lexicon expansion. In *SemEval'13*, 2013.
[10] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.
[11] K. Chakrabarti et al. Approximate query processing using wavelets. *VLDB*, 2001.
[12] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 2007.
[13] A. Gattani et al. Entity extraction, linking, classification, and tagging for social media: A wikipedia-based approach. *Proc. VLDB Endow.*, 6(11), 2013.
[14] A. C. Gilbert et al. Optimal and approximate computation of summary statistics for range aggregates. PODS, 2001.
[15] J. M. Hellerstein et al. Online aggregation. *SIGMOD*, page 171–182, June 1997.
[16] A. Holub, P. Perona, and M. C. Burl. Entropy-based active learning for object recognition. In *CVPR Workshops*, pages 1–8. IEEE Computer Society, 2008.
[17] R.-L. Hsu et al. Face detection in color images. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 2002.
[18] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679 – 688, 2006.
[19] P. JACCARD. Etude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.
[20] M. Joglekar et al. Exploiting correlations for expensive predicate evaluation. SIGMOD '15, New York, NY, USA, 2015. ACM.
[21] K. Karanasos et al. Dynamically optimizing queries over large scale data platforms. SIGMOD, 2014.
[22] C. Koch et al. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.
[23] M. Lapin et al. Top-k multiclass SVM. In *NIPS 2015*.
[24] I. Lazaridis et al. Optimization of multi-version expensive predicates. SIGMOD'07.
[25] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.
[26] J. Li and A. Deshpande. Consensus answers for queries over probabilistic databases. In *PODS*, pages 259–268. ACM, 2009.
[27] Y. Lu et al. Accelerating machine learning inference with probabilistic predicates. SIGMOD '18, New York, NY, USA, 2018. ACM.
[28] V. Markl et al. Robust query processing through progressive optimization. SIGMOD '04.
[29] D. Marmaros et al. Pay-as-you-go entity resolution. *IEEE TKDE*, 2013.
[30] K. Mikolajczyk et al. Human detection based on a probabilistic assembly of robust part detectors. In *ECCV 2004*.
[31] U. F. Minhas and A. Kumar. SIGMOD 2021 curated session: Data management for ML. https://2021.sigmod.org/program/program_tuesday.shtml.
[32] M. Nikolic et al. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264. ACM, 2014.
[33] R. Nuray-Turan, D. V. Kalashnikov, S. Mehrotra, and Y. Yu. Attribute and object selection queries on objects with probabilistic attributes. *ACM Trans. Database Syst.*, 37(1):3:1–3:41, 2012.
[34] R. Olfati-Saber et al. Consensus filters for sensor networks and distributed sensor fusion. CDC '05, Dec 2005.
[35] T. Papenbrock et al. Progressive duplicate detection. *IEEE TKDE*, 2015.
[36] Y. Park et al. Verdictdb: Universalizing approximate query processing. SIGMOD'18.
[37] J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*.
[38] D. Powers et al. Evaluation: From precision, recall and f-measure to roc, informedness, markedness & correlation. *J. Mach. Learn. Technol*, 2:2229–3981, 01 2011.
[39] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. VLDB, 2001.
[40] N. F. F. D. Silva et al. A survey and comparative study of tweet sentiment analysis via semi-supervised learning. *ACM Comput. Surv.*, 2016.
[41] T. Sim et al. The cmu pose, illumination, and expression (pie) database. In *Int. Conf. on Automatic Face Gesture Recognition*, 2002.
[42] S. Singh et al. Indexing uncertain categorical data. In *ICDE*, pages 616–625, 2007.
[43] J. A. K. Suykens et al. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, 1999.
[44] C. J. Willmott and K. Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.
[45] W. Wu, Naughton, et al. Sampling-based query re-optimization. SIGMOD '16.
[46] J. Xu, D. V. Kalashnikov, and S. Mehrotra. Query aware determinization of uncertain objects. *IEEE Trans. Knowl. Data Eng.*, 27(1):207–221, 2015.
[47] B. Zadrozny and C. Elkan. Transforming classifier scores into accurate multiclass probability estimates. KDD, 2002.

6.1 Exploiting Joins in Probe Query

The steps for exploiting join conditions on fixed attributes for generating `probe queries` are as follows:

[Step 1]: *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). For example, Figure 5b shows the query tree generated from the query of Figure 5a. 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 2]: *Generating Join Graph*: This step and the next step 3 are performed to exploit the join conditions on fixed attributes in a query to filter out tuples of R_i that do not require enrichment. Given a query tree with selection conditions modified as in Step 2, 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 join graph, the nodes correspond to *reduced* relations, *i.e.*, relations with the selection conditions applied on them. If there exists a join condition between the two relations in the original query, an edge between two nodes is present and shows the join conditions between two relations expressed in CNF form.

Next, from each edge of the join graph, all the derived join conditions are removed. If after removing all derived conditions of a join node, the final condition becomes empty (*i.e.*, all the conjuncts were on derived attributes), then that edge is deleted from the graph, *i.e.*, none of the join conditions between the two relations can be exploited to reduce the set of tuples that require enrichment.[8]

E.g., in Figure 6a, we present a join-graph for the query tree shown in Figure 5b. This graph contains two nodes: $\langle N_1, N_2\rangle$, representing the reduced relations of $\langle R_1, R_2\rangle$, respectively, *i.e.*, after applying selection conditions on derived attributes of each relation. Here, the edge between N_1 and N_2 represents the join condition of $(R_1.A_4 = R_2.A_4)$ (after removing the join condition on $R_1.\mathcal{A}_3 = R_2.\mathcal{A}_3$) from Figure 5b).

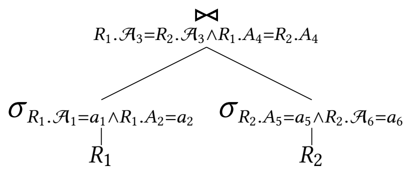
[Step 3]: *Semi-join-based Reduction*: Given the join graph as an input, for each node N_i in the graph, this step generates a set of semi-join programs for N_i to reduce the number of tuples of N_i that require enrichment. For N_i, semi-join programs are generated by exploiting join conditions among nodes of the graph. For a 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 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

[8] If a query tree contains the operators of `union`, `set-difference`, or `cross product`, then they are ignored, as such operators can not be utilized to reduce the number of tuples in `probe queries` apart from the join conditions.

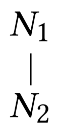
SELECT * FROM R_1, R_2 WHERE $R_1.\mathcal{A}_1 = a_1$ AND $R_1.A_2 = a_2$
AND $R_1.\mathcal{A}_3 = R_2.\mathcal{A}_3$ AND $R_1.A_4 = R_2.A_4$
AND $R_2.A_5 = a_5$ AND $R_2.\mathcal{A}_6 = a_6$

(a) Original query.


$\bowtie_{R_1.\mathcal{A}_3=R_2.\mathcal{A}_3 \wedge R_1.A_4=R_2.A_4}$
$\sigma_{R_1.\mathcal{A}_1=a_1 \wedge R_1.A_2=a_2}$
$\sigma_{R_2.A_5=a_5 \wedge R_2.\mathcal{A}_6=a_6}$
R_1
R_2

(b) Query tree.

Figure 5: Original query and query tree for probe query generation.



(a) Join graph (Node $N_1 : \sigma_{R_1.A_2=a_2}(R_1)$ and Node $N_2 : \sigma_{R_2.A_5=a_5}R_2$).

Semi-join Programs for R_1:
$\langle N_1 \ltimes_{R_1.A_4=R_2.A_4} N_2; \rangle$

The query that filters out tuples of R_1 that do not join with any tuples of R_2:
SELECT * FROM R_1 WHERE $R_1.A_2 = a_2$ AND $R_1.A_4$ IN (SELECT A_4 FROM R_2 WHERE $R_2.A_5 = a_5$)

(b) Semi-join program and the query to filter out tuples of R_1 based on join conditions on fixed attributes (step 2 of `probe query` generation for R_1).

Figure 6: The join graph of R_1 and the semijoin program used to filter out tuples of R_1 in the `probe query` of R_1 (for the original query of Figure 5(a)).

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 (same as the graph shown in Figure 6a). In $ST(N_1)$, a semi-join between R_1 and the tuples of R_2 are performed based on the join condition of $R_1.A_4 = R_2.A_4$. Using this semi-join programs, this step is able to eliminate the tuples of R_1 that do not join with any tuple of R_2. This step for semi-join reduction we used is based on the seminal work on semi-join given in [10].

[Step 4]: ***Generating queries from semi-join program***: Given the semi-join programs generated an input, for each relation $R_i \in q$, this step generates a query based on the semi-join programs and the selection conditions on R_i in a straightforward manner. For example, in Figure 6b (bottom), we show the `probe query` generated for R_1, from the semi-join programs described in 6b (top) for R_1. All the tuples of R_1 that do not match with any possible values of $R_2.A_4$ are filtered out using this query.

6.2 Threshold Selection

Theorem 2. *Let L be a list of tuples that can be part of the query answer and are sorted in a decreasing order of their probability of satisfying all the conditions on derived attributes of a query Q. Let L^k be a list of tuples consisting of only first k tuples from the sorted list of L. The expected quality of the possible subsets of L follow a monotonically increasing pattern with respect to k, up to a certain value of k and beyond that it decreases monotonically, i.e., it follows the pattern: $E(Qty(L^1)) < E(Qty(L^2)) < \ldots < E(Qty(L^{\tau-1})) > E(Qty(L^{\tau})) > E(Qty(L^{\tau+1})) > \ldots > E(Qty(L))$.*

Proof. We prove this theorem using F_α-measure as the quality metric of the answer set and show that if $E(F_\alpha)$ decreases for the first time due to the inclusion of a tuple in Ans_w, then it keeps decreasing monotonically with the inclusion of any further tuples. We denote $E(F_\alpha)$ of Ans_w, if τ-th tuple is included in Ans_w as F_τ. Similarly, the $E(F_\alpha)$ measures corresponding to the inclusion of $\tau+1$-th and $\tau+2$-th tuple are denoted as $F_{\tau+1}$ and $F_{\tau+2}$ respectively. We show that for a particular value of τ, if $F_{\tau+1} < F_\tau$, then it implies $F_{\tau+2} < F_{\tau+1}$.

$$F_\tau = \frac{(1+\alpha).\frac{k_1}{\tau}\cdot\frac{k_1}{k_2}}{\alpha\cdot\frac{k_1}{\tau}+\frac{k_1}{k_2}} = \frac{(1+\alpha)\cdot k_1}{\alpha\cdot k_2+\tau}, k_1 = \sum_{i=1}^{\tau}\mathcal{P}_i, k_2 = \sum_{i=1}^{|Ans_w^{MAX}|}\mathcal{P}_i \tag{15}$$

where $\mathcal{P}_i$ is the probability of a tuple satisfying all the conditions of Q and Ans_w^{MAX} is the set of tuples that have non-zero probability of being part of the query result.

Similarly the values of $F_{\tau+1}$ and $F_{\tau+2}$ are as follows:

$$F_{\tau+1} = \frac{(1+\alpha)(k_1+\mathcal{P}_{\tau+1})}{(\alpha k_2+\tau+1)}, F_{\tau+2} = \frac{(1+\alpha)(k_1+\mathcal{P}_{\tau+1}+\mathcal{P}_{\tau+2})}{(\alpha k_2+\tau+2)} \tag{16}$$

$$\begin{aligned} F_{\tau+1} < F_\tau &\Rightarrow \frac{(1+\alpha)\cdot(k_1+\mathcal{P}_{\tau+1})}{(\alpha k_2+\tau+1)} < \frac{(1+\alpha)\cdot k_1}{\alpha k_2+\tau} \\ &\Rightarrow (k_1+\mathcal{P}_{\tau+1})(\alpha k_2+\tau) < k_1(\alpha k_2+\tau+1) \\ &\Rightarrow \alpha k_1 k_2 + k_1\tau + \alpha k_2\mathcal{P}_{\tau+1} + \tau\mathcal{P}_{\tau+1} < \alpha k_1 k_2 + k_1\tau + k_1 \end{aligned} \tag{17}$$

Simplifying the above, we derive the following condition: $\frac{(k_1+\mathcal{P}_{\tau+1}+\mathcal{P}_{\tau+2})}{(\alpha k_2+\tau+2)} < \frac{(k_1+\mathcal{P}_{\tau+1})}{\alpha k_2+\tau+1}$, *i.e.*, $F_{\tau+2} < F_{\tau+1}$. □

6.3 Choosing Tuples

For each relation $R_i \in Q$, the tuples that did not contribute to the result of previous epoch, are considered for enrichment. Such tuples have higher probability of improving the quality of the query result. Below, we formalize this observation using a theorem.

Theorem 3. *Enriching a tuple t_k of a relation R_i that did not contribute to any tuple of the answer in epoch e_{w-1}, ensures that the quality of the answer set increases in epoch e_w as compared to the previous epoch of e_{w-1}. That is, $E(Qty(Ans_w)) >= E(Qty(Ans_{w-1}))$ irrespective of the outcome of enrichment on the tuple.*

Proof. We prove this theorem using the following lemmas.

Lemma 1. *If the probability of a tuple $t_k \in R_i$ that contributed in Ans_{w-1} increases in epoch e_w, then the $E(F_\alpha)$ measure of Ans_w can increase or decrease from the result of previous epoch, i.e., Ans_{w-1}. If the probability decreases, then $E(F_\alpha)$ measure of Ans_w always decreases.*

If the probability of t_k increases from the previous epoch of e_{w-1}, it results in increment of the probability of the answer tuples (*i.e.*, in Ans_{w-1}) that were generated by t_k. Let the increase in sum of the probability values of the tuples that were part of Ans_{w-1} be Δ_1 and the increase in the summation of probabilities that were outside of Ans_{w-1} be Δ_2. Considering the expression of $\hat{F}_\alpha(Ans_w)$ in Equation 5, the numerator increases by the amount of Δ_1, whereas the denominator increases by a greater amount of Δ_2. Hence, the expected quality of $\hat{F}_\alpha(Ans_w)$ decreases from $\hat{F}_\alpha(Ans_{w-1})$. However, if some more tuples are added to Ans_{w-1} based on the new answer-threshold of e_w, then the value of Δ_1 can be higher than Δ_2, resulting in an increment of $F_\alpha(Ans_w)$. Similarly, when the probability of t_k decreases from previous epoch, the summation of probability values of the tuples that were part of Ans_{w-1} decreases resulting in a decrement of $\hat{F}_\alpha(Ans_w)$ value. ■

Lemma 2. *If the probability of a tuple $t_k \in R_i$ that did not contribute to any tuple in Ans_{w-1} increases or decreases in epoch e_w, then the $E(F_\alpha)$ measure of Ans_w will always be higher than Ans_{w-1}.*

If the probability of t_k increases from the previous epoch of e_{w-1}, it results in increment of the probability of the tuples that were generated by t_k and were not part of Ans_w. Let the sum of the probability values of the tuples that have probability higher than the answer threshold of epoch e_{w-1} be Δ_1 and the increase in the summation of probabilities that are still outside of Ans_{w-1} be Δ_2. Note that since the value of Δ_1 is the summation of the probability of all the newly added tuples to the answer set, the value of Δ_1 is much higher than the sum of delta values *i.e.*, Δ_2. Hence, considering the expression of $\hat{F}_\alpha(Ans_w)$ in Equation 5, the numerator increases by a greater amount of Δ_1, as compared to a lower amount of Δ_2 in the denominator. Hence, the expected quality of $\hat{F}_\alpha(Ans_w)$ increases from $\hat{F}_\alpha(Ans_{w-1})$ when t_k is enriched. ■