JENNER: Just-in-time Enrichment in Query Processing

Dhrubajyoti Ghosh
University of California, Irvine
USA
dhrubajg@uci.edu

Peeyush Gupta
University of California, Irvine
USA
peeyushg@uci.edu

Sharad Mehrotra
University of California, Irvine
USA
sharad@ics.uci.edu

Roberto Yus
University of Maryland, Baltimore County
USA
ryus@umbc.edu

Yasser Altowim
Saudi Data and Artificial Intelligence Authority
Saudi Arabia
ytowim@nic.gov.sa

ABSTRACT

Emerging domains, such as sensor-driven smart spaces and social media analytics, require incoming data to be *enriched* prior to its use. Enrichment often consists of machine learning (ML) functions that are too expensive/infeasible to execute at ingestion. We develop a strategy entitled Just-in-time ENrichmeNt in quERy Processing (JENNER) to support interactive analytics over data as soon as it arrives for such application context. JENNER exploits the inherent tradeoffs of cost and quality often displayed by the ML functions to progressively improve query answers during query execution. We describe how JENNER works for a large class of SPJ and aggregation queries that form the bulk of data analytics workload. Our experimental results on real datasets (IoT and Tweet) show that JENNER achieves progressive answers performing significantly better than the naive strategies of achieving progressive computation.

PVLDB Reference Format:
Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, Roberto Yus, and Yasser Altowim. JENNER: Just-in-time Enrichment in Query Processing . PVLDB, 15(11): 2666 - 2678, 2022.
doi:10.14778/3551793.3551822

PVLDB Artifact Availability:
The full version of paper, source code, and data have been made available at https://github.com/jennerrepo/jenner.

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 [1]. Often, such data needs to be enriched before it can be analyzed. Functions used to enrich data (referred to as *enrichment functions* in the paper) could consist of (a combination of) custom-compiled code, declarative queries, and/or expensive machine learning techniques. Examples include mechanisms for sentiment analysis [54] over social media posts, named entity extraction [7] in text, and sensor interpretation such as face detection and face recognition [32, 43] from images, sensor data fusion [46], and data cleaning tasks such as missing value imputation in relational data [15].

Analytical applications that require raw data to be enriched prior to use can be built in several ways. One approach is to collect the data, periodically enrich it, and then load it into the database for analysis, as done in the traditional extract-transform-load (ETL) systems [56]. An alternate approach is to enrich the data as it arrives "on-the-fly" at insertion time. Systems (*e.g.*, Spark Streaming [62] often used for scalable ingestion) are capable of executing enrichment functions on newly arriving data prior to its storage in a DBMS and can be used to build such an approach.

Both approaches suffer from several limitations. The periodic approach could significantly increase latency between when data arrives to when it is enriched and available for analysis. It does not support interactive analytics on the data as it is inserted.[1] The alternate approach of enriching data at insertion time suffers from a different limitation - it is only feasible when enrichment functions are simple. If complex functions such as Multi-layer Perceptron and Random Forest (often used to interpret data) are used for enrichment, it may result in ingestion bottleneck. For instance, accurately locating a person using WiFi events based on their history of connections can take ≈ 200ms/event [38, 39]. Clearly, with thousands of connection events with WiFi access points per second, it is infeasible to execute such functions at the ingestion time.

Both the insertion time and periodic enrichment approach waste resources as they indiscriminately enrich all data, irrespective of whether they are needed for future analysis. *E.g.*, in the previous example, the analyst may only be interested in executing ad-hoc queries that require fine-grained localization of a small number of individuals instead of continuous fine-grained location of all individuals. Finally, both approaches require that enrichment functions to be known in advance prior to data collection.

With the goal of supporting analytics on data as it arrives, recent work in EnrichDB [2] has proposed a different approach that supports "just-in-time" data enrichment during query processing. Such a technique eliminates several limitations of the insertion-time and periodic techniques mentioned above. It prevents wastage of resources since only data accessed by queries during analysis is enriched. Furthermore, by eliminating enrichment at ingestion

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551822

[1] This is similar to the limitation of the traditional data warehouse applications wherein newly arrived data was not available for analysis until it was loaded into a warehouse which led to the emergence of HTAP systems [12] over the past decade.

(or limiting it to simple enrichment) the scheme does not create an ingestion bottleneck. Finally, the data is available for analysis as soon as it is inserted without incurring long latency between when data is inserted and is available for analysis.

While the query-time enrichment has several advantages, it nonetheless, increases latency of individual queries since enrichment of queried data need to be performed at the time of query execution. Interactive analytics can only be supported if the number of objects that require enrichment are small. To overcome such a limitation of query-time enrichment, we develop Just-in-time ENrichmeNt in quERy Processing (JENNER) that interactively refines query answers based on progressively enriching data. JENNER exploits the fact that often multiple functions can be used to enrich the data for the same task which display a cost-quality tradeoff.

Cost-quality tradeoffs are demonstrated by several ML functions, *e.g.*, a complex neural network classifier (with higher number of layers and/or higher number of neurons per layer) has better accuracy than a neural network with lower complexity.[2] Furthermore, simple classifiers such as Bayes classifier, logistic regression are less accurate than complex classifiers of Extra Tree, Random Forest, and Neural Network classifiers. Such tradeoff is also studied in LOCATER [38] where coarse level localization techniques were used which had a fraction of cost than the fine-grained techniques.

JENNER prioritizes/ranks which enrichment functions should be evaluated on which objects to improve the quality of the answer as quickly as possible for a query. JENNER divides the query execution time into several *epochs* and analyzes the evaluation progress at the beginning of each epoch to generate an enrichment plan. Such a plan includes tuples that have highest potential of improving the quality of the answer in that epoch.

The deferred/lazy enrichment in JENNER is motivated by prior work on lazy query time data cleaning such as [11, 26]. Such works 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, [8] developed an approach to dynamically link entities in top-k queries. Such approaches, however, do not support the progressiveness that JENNER does. Since data cleaning tasks, such as entity linking, can be viewed as enrichment, the approach developed in this paper can also benefit query time data cleaning by supporting progressiveness.

In summary, we propose JENNER, a progressive approach of answering queries on the data that needs to be enriched. JENNER orders enrichment in a way to optimize progressiveness of queries by using a probabilistic strategy of estimating the benefit of enrichment functions. Our experiments on real data and enrichment functions show the efficacy of JENNER. JENNER has been integrated into the EnrichDB system [3] and forms the basis of its progressive query evaluation strategy.

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 (by executing a set of enrichment functions). The remaining attributes are ***fixed*** (denoted as A_j) and do not require enrichment. Table 1 shows a relation with fixed attributes (*e.g.*, `id`, `user_id`, `time`, `wifi_ap`) and a derived attribute of `location` that could be derived by multiple functions that vary in cost and quality.

[2]This phenomena is true until the model overfits the training data [22].

The enrichment functions are 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. The output of both types of enrichment functions are for a single derived attribute of a tuple. A single-tuple-input enrichment function typically uses other attributes in the same tuple and a model to make the inference. The multi-tuple-input enrichment function takes the fixed attribute values of multiple tuples from the same relation or different relations based on a parameter to infer the derived attribute value of the tuple.

Enrichment functions can also be categorized by their output types: (*i*) *single-valued*, (*ii*) *multi-valued*, or (*iii*) *probabilistic*, that output as a prediction a single value, multiple values, or probability distribution over a set of possible values. Of the three, probabilistic outputs are most general as we can always interpret results of the other two as probability distributions. We, thus, assume that enrichment functions output probabilities in the rest of the paper. *E.g.*, the value of `location` in tuple t_1 of `wifi` in Table 2 is $[0.54, 0.35, 0.11]$ which corresponds to locations `L1`, `L2`, and `L3`. JENNER assumes that the enrichment functions are calibrated using mechanisms such as [49, 61] on a labeled validation dataset. After calibration, the enrichment functions output a real probability distribution.

The enrichment functions for a derived attribute $\mathcal{A}_i$ are denoted by $F^i = \{f_1^i, f_2^i, \ldots, f_k^i\}$. Each function f_j^i is associated with a *cost* (denoted by c_j^i) which represents the average execution cost of the function on a single tuple. Since the output of an enrichment function is probabilistic, we 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 measure uncertainty using the entropy metric [31]. For a tuple t_k and derived attribute $\mathcal{A}_p$, entropy is calculated as follows:

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

where, p_i represents the probability of the derived attribute taking the i-th domain value for the tuple t_k. The entropy of t_1 in Table 2 is $[-0.54\times\log_3(0.54) - 0.35\times\log_3(0.35) - 0.11\times\log_3(0.11)] = 0.86$.

JENNER is based on the premise that enrichment functions (that are often based on ML models) can be ordered based on their average behavior and very often enrichment functions that are more accurate are more expensive. This behavior of classifiers was highlighted in the past, *e.g.*, [21] studied cost-accuracy tradeoff of classifiers for sentiment analysis in tweets. Other examples include work on deep neural networks that accelerated performance by reducing the floating point precision of intermediate outputs of the neural network [25, 53, 63] and reducing the network size by modifying the width of the layers or by skipping layers/modules [14, 57, 60]. Such techniques trade between complexity and cost, *e.g.*, [63] describes a "precision-adjustable" softmax computation to achieve different precision and cost requirements of ML tasks.

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 their output. The state has two

Table 1: The `wifi` table where `location` is a derived attribute.

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

Table 2: State output for derived attributes.

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

Table 3: `wifistate` table (created for tuples in `wifi` table).

tid	BitMap	LocationState Output
t_1	[1,0,0]	[[0.54, 0.35, 0.11, ...], [], []]
t_2	[1,0,1]	[[0.2,0.6,..., 0.2],[], [0.86,0.1, . . . 0.04]]
t_3	[1,1,0]	[[0.1,0.2,...,0.5,0.2], [0.2,0.5,0,...] ,[]]

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 $t_i.\mathcal{A}_j$. *E.g.*, considering three enrichment functions f_1, f_2, and f_3 for `location`, the state bitmap of t_2, *i.e.*, $\langle 101\rangle$ signifies that only f_1 and f_3 have been executed on it (see Table 3).[3] Further, the output of the state $\langle[0.2, 0.6, \ldots 0.2], [], [0.86, 0.1, \ldots, 0.04]\rangle$ is the output of f_1 and f_3.

The individual function outputs are aggregated into a combined value denoted by $\mathcal{A}_j.Value$ (*e.g.*, *Location.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 particular value a_j by $Val(state(t_i.\mathcal{A}_j))[a_j]$. *E.g.*, in Table 3, the value of `location` for t_1, is $Val(state(t_1.\texttt{location}))$ = [L1: 0.54, L2: 0.35, L2: 0.11]. The probability of t_1 taking the value of `location` as L1, *i.e.*, $Val(state(t_1.\texttt{location}))[\texttt{L1}]$ is 0.54.

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 provided by the user as a preprocessing step. The size of the validation dataset is small and can be chosen from the same training dataset on which the enrichment functions are trained. 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 next-best function, denoted as $NBF(t_i, \mathcal{A}_j)$. Note that uncertainty reduction due to enrichment functions that have already been executed in the past is zero and hence they cannot be the next best function at a state.

Example 2.1. Suppose the value of the *location* attribute of a tuple t_1 is: [L1: 0.54, L2: 0.35, L3: 0.11]. The entropy of t_1 with respect to *location* is 0.86 (measured using Equation 1). Suppose after the enrichment of t_1 and using the next best function and combining the output with the outputs of previously executed functions, the *location* value of t_1 becomes [L1: 0.8, L2: 0.15, L3: 0.05]. Hence, the entropy of the tuple with respect to *location* reduces to 0.56.

Currently JENNER rank orders enrichment functions based on their capability to reduce uncertainty (on an average) of derived attributes. Prior work [16] showed that ranking of classifiers can be context dependent. In [16], authors developed a strategy to train multiple classifiers where each classifier was an expert on a part of the data and their outputs were combined using a stacking based method. Such techniques result in the ranking to be context dependent. They can be supported in JENNER by storing a map that orders functions based on the context instead of a single next-best function. JENNER needs to store the condition on context (*e.g.*, `timestamp` between t_1 and t_2) as the key and next best function as the value. When $NBF(t_i, \mathcal{A}_j)$ is called, JENNER will use the context of t_i to find next-best function. Implementing JENNER with context dependent enrichment function is an interesting future direction.

[3] The bitmap does not represent any order between enrichment functions. It is possible to execute only the second enrichment function without executing the first one.

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 wifi.location as p_location, wifi.time as
    p_time  FROM wifi WHERE p_location = 'L1' AND
    p_time BETWEEN ('10:00','12:00')
```

Code Listing 1: Example Query.

In the following, we focus on the queries with at least one derived attribute in the SELECT or WHERE clause. *E.g.*, the above query contains a condition on the derived attribute `location`.

Since derived attributes have probabilistic values, JENNER interprets queries based on ***determinization-based*** semantics [37, 45, 59], wherein Q is evaluated over determinized values for derived attributes in tuples that are part of Q. The determinized value of a derived attribute $\mathcal{A}_j$ 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 [37, 45, 59]. We choose the determinization function that returns the highest probable value after combining the output of the previously executed enrichment functions on the tuple. If the highest probable value is not unique then JENNER assigns the attribute value as NULL. For multi-valued determinization functions such as determinization based on a threshold, if all the possible attribute values have the probability lower than the threshold for a particular tuple, then the derived attribute is assigned a NULL value for that tuple.[4]

Progressive Query Execution in Epochs. 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 $|e_w|$.[5]

[4] The techniques developed in the paper can be adopted to other determinization functions studied in [37, 45, 59] including those that return multiple values as discussed in the extended version of the paper [4].

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.

[5] For simplicity, we will consider the duration of each epoch to be of fixed size in the remainder of the paper, though, the approach does not require this to be the case.

During an epoch e_w, certain enrichment functions (not executed before) are chosen to be executed. Let $EP_w = \{\langle t_i, \mathcal{A}_j, f_k^j \rangle\}$ be a set of ⟨tuple, derived attribute, enrichment function⟩ triples, referred to as the *enrichment plan* of epoch e_w. Let S be the state of the database at the end of e_{w-1} and, let S' be the state after the execution of the enrichment plan EP_w in e_w. This results in state update of all 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 as: $state(t_i.\mathcal{A}_j).output = (t_i.\mathcal{A}_j).output \oplus \langle t_i, \mathcal{A}_j, f_k^j \rangle$, where $\oplus$ signifies that the k-th array of state output is updated and the new derived attribute value of the tuple is updated. At the end of e_w, user receives a query result, denoted as Ans_w, based on the current state of the database. Note that a tuple that was 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 data in different epochs, we will denote the original database D on which Q executes as D_0 to signify its status prior to the query execution. We will refer to the database after the execution of e_w as D_w that corresponds to the database after all the enrichment functions until epoch e_w have been executed.

The user can access the query results at the end of each epoch. JENNER provides an expected quality of the returned results based on the enrichment performed until that time. This quality can be used by the user to determine if the query execution needs to be continued. The progressive query execution is motivated by online AQP systems where users can view the query results as soon as they are computed on the samples [17, 30]. The usefulness of online AQP systems were discussed in [24, 44]. An alternative is to use the query model of offline AQP systems [6, 48] where a maximum duration/quality requirement is specified by the user. The system continues the query execution till that time. JENNER supports the model of offline AQP by appropriately setting the epoch sizes.

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 used in [9, 47]):

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

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, $W(e_{i-1}) > W(e_i)$), 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. Assigning higher weights to the earlier epochs provide higher importance to the improvement in quality in the earlier epochs. ■

Since weights W_i in the progressive score defined above are decreasing, optimizing the progressive 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 [50], or Jaccard similarity coefficient [34]. We define the F_α measure 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 is precision, *i.e.*, $Pre(Ans_w) = |Ans_w \cap Ans^{real}|/|Ans_w|$, and Rec is recall, *i.e.*, $Rec(Ans_w) = |Ans_w \cap Ans^{real}|/|Ans^{real}|$, and $\alpha \in [0,1]$ is the weight of precision in F_α-measure. In the rest of the paper, for computing the quality of set-based query result, we restrict to F_α-measure. The quality of an aggregation query could be measured using root-mean-square error [33] or mean-absolute-error [58].

Quality Guarantees in JENNER. At each epoch JENNER strives to choose the best set of objects to enrich that can improve the quality of the query result most. Since the ground truth of objects are not known, JENNER can neither directly measure the quality of query 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 (of 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 maximizes the improvement in quality of the resulting query answer from previous epoch.

Below we discuss how JENNER estimates the quality for set-based queries. For aggregation queries, JENNER optimizes the enrichment process using a set-based metric (*e.g.*, F_α-measure) and then applies the aggregation function on the resulting tuples.

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 compute the estimated precision and recall of Ans_w, denoted by $\widehat{Pre}$ and $\widehat{Rec}$, as follows:

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

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

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

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 of the query. For a selection query with a single condition ($t_i.\mathcal{A}_j = a_j$) on a derived attribute $\mathcal{A}_j$, the $\mathcal{P}_i$ 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 queries with selection conditions on multiple derived attributes, the probability of t_i satisfying the predicate is computed under the independence assumption of derived attributes by combining the probabilities of t_i satisfying the predicates on single attributes. For a join query, $\mathcal{P}_i$ of a tuple t_i in a base relation is calculated as follows: (*i*) for each answer tuple that was generated from t_i, the probability of the answer tuple satisfying all the conditions on derived attributes are combined according to the independence assumption and (*ii*) the probability of all the answer tuples generated from t_i are added to generate $\mathcal{P}_i$ as done in [55]. JENNER uses expected quality metric to choose which tuples should be enriched in an epoch as will be clear in §3. Note that it is possible that after enriching a tuple t_i, the probability $\mathcal{P}_i$ of the tuple to satisfy the query decreases from the previous epoch, but it is expected that the overall quality of the query result improves. Furthermore, if multiple enrichment functions with cost quality tradeoff can not be defined (*e.g.*, enrichment can only be performed by a unique function), JENNER may not achieve progressive improvement of query results. Even in such a case, as JENNER only enriches objects relevant to the query, it is beneficial than complete enrichment of objects before answering queries.

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 epoch e_{w-1}, where $w \leq n-1$. Let CS_w be the set of tuples in D that are not fully enriched. The progressive enrichment problem consists of determining a set of ⟨tuple, derived attribute, enrichment function⟩ triples EP_w such that, when executed in e_w, results in a database value of: $(Val(state(D)) \oplus EP_w)$ that optimizes the following objective:

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

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

The overall algorithm of JENNER is presented in Algorithm 1. The zero-th epoch performs pre-processing in order for the answers to be generated in the later epochs. The later epochs of e_w, $w > 0$, iteratively enriches the tuples and compute the query results.

Zero-th Epoch (Lines 5 - 10): The goal of the zero-th epoch is to seed JENNER with the tuples that may need to be enriched in the upcoming epochs (*i.e.*, epoch 1 and onwards). It also sets up the data structures used in the later epochs. JENNER does not require any enrichment function to be executed on the data before. In zero-th epoch, 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* (Line 4). 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

Algorithm 1: Overall Algorithm.

Inputs: Query Q and the duration of each epoch *epoch_duration*.
Outputs: An enrichment plan for each epoch.
1 Function *Optimize_Enrichment*() **begin**
2 $CandidateSet^M \leftarrow \emptyset$
3 **Epoch 0:: for** each $R_i \in Q$ **do**
4 $CandidateSet(R_i) \leftarrow Execute(GenerateProbeQ(Q, R_i))$
5 **for** each $R_i.\mathcal{A}_k \in Q$ **do**
6 **for** each $t_j \in CandidateSet(R_i)$ **do**
7 $M \leftarrow CompProb(t_j, \mathcal{A}_k)$; $C \leftarrow Cost(NBF(t_j, \mathcal{A}_k))$
8 $B \leftarrow ComputeBenefit(NBF(t_j, \mathcal{A}_k))$
9 $CandidateSet^M \cup \langle t_j, \mathcal{A}_k, NBF, B, C, M \rangle$
10 $CandidateSet^M \leftarrow Sort_{match\_prob}(CandidateSet^M)$
11 **Epoch** w, $w \geq 1$**:: for** each epoch e_w **do**
12 $EP_w \leftarrow ChooseEnrichmentPlan(CandidateSet^M)$
13 **for** each entry $\in EP_w$ **do**
14 $ExecuteEnrichment(t, \mathcal{A}, f)$; $UpdateState(t, \mathcal{A}, f)$;
15 $Determinize(t, \mathcal{A})$;$UpdateBenefit(CandidateSet^M, t, \mathcal{A})$;
16 $Ans_w \leftarrow ProduceQueryResult(Q)$
17 Return *Ans*

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 7- 8).

The *match_probability* of a derived attribute $t_i.\mathcal{A}_j$ is determined using the probability $Val(state(t_i.\mathcal{A}_j))[a_j]$ 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.

The benefit of enrichment of tuple t_i using the next best function (as shown in Line 8) 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 (Lines 7-9). Candidates in $CandidateSet^M$ are sorted based on their *match_prob* values (Line 10).

Later epochs $e_w, w \geq 1$, (Lines 11 - 16): 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 the epoch e_w (Line 12); 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 14 - 15); We discuss them 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 $E(F_\alpha)$ measure (Line 16); We discuss this step in §3.5. Progressive approach for aggregation queries is realized by developing a progressive 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 the query results, one could add all the tuples that are not fully enriched to this set. However, it would result in a significant number of redundant enrichments, *i.e.*, the tuples that do not satisfy predicates on the

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

fixed attributes will be added to this set for enrichment. To avoid this, JENNER exploits the predicates over fixed attributes to filter out tuples whose enrichment has no consequence on the results. 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 $pq(R_i)$).

We illustrate how probe queries are generated using the query in Figure 1a. The selection conditions on fixed attributes of R_1 are identified, *i.e.*, $R_1.A_2 = a_2$. The tuples of R_1 that require enrichment are restricted using this condition as shown in Figure 1b. JENNER further exploits join conditions on fixed attributes. *E.g.*, in Figure 1a, an R_1 tuple must join with at-least one of R_2 tuples that satisfy the condition of $R_2.A_5 = a_5$. A tuple of R_1 can be part of the answer if it joins some tuples of R_2 which satisfy the join condition $R_1.A_4 = R_2.A_4$. JENNER determines such a set by computing semi-join with other relations in the query with which R_1 joins using conditions on fixed attributes. Utilizing the semi-join optimization results in a nested query as shown in Figure 1c. We restrict the description of probe query using an example as the algorithm is an adaptation of semi-join optimization in [13]. The algorithm is available in [4].

Further, JENNER exploits the current state of the tuples to avoid repeated enrichment of tuples that are completely enriched. JENNER rewrites the selection condition as in Figure 1d on derived attribute, *i.e.*, $R_1.\mathcal{A}_1 = a_1$ by the condition: $\left[R_1.id = R_1State.id \text{ AND } R_1.array\_sum(\mathcal{A}_1 StateBitmap)! = R_1.array\_length(\mathcal{A}_1 StateBitmap)\right]$. This checks if a derived attribute is completely enriched using *StateBitmap* column of state table (*i.e.*, all the bits set to 1). Such tuples are not enriched.

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 from previous epoch due to the execution of the enrichment plan. We restrict the choice of tuples for enrichment to only

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 in e_{w-1}.
Outputs: The benefit of the ⟨tuple t_i, derived attribute $\mathcal{A}_j$, enrichment function f_k⟩ triplet.

```
Function Compute_Benefit() begin
    PrevQuality ← F̂_α(Ans_{w-1})
    E_{w-1} ← ComputeEntropy(t_i, A_j)
    Ê_w ← E_{w-1} − DeltaEntropy(t_i, A_j, f_k)
    Match_Probability ← ComputeInverseOfEntropy(Ê_w)
    ExpectedQuality ← Quality(Match_Probability, Ans_{w-1})
    Benefit(t_i, A_j, f_k) ← Max(ExpectedQuality − PrevQuality, 0)
    Return Benefit(t_i, A_j, f_k)
```

those that are not in the answer set of previous epoch. This step is performed as the benefit of further enriching a tuple in e_w that was in Ans_{w-1} is significantly lower than the tuples that are not in Ans_{w-1}. We formally justify this decision 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 e_w. The benefit of $\langle t_i, \mathcal{A}_j, f_k\rangle$ is defined as follows:

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

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, JENNER estimates (*i*) the quality of answer after epoch e_{w-1} and (*ii*) the expected quality of the answer set if the enrichment function is executed in the current epoch.

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 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 `wifi` (see Table 1). Suppose at the end of epoch e_1, the state of the tuples are as shown in Table 3 and let t_1 be part of the query result. Since, the location of t_1, *i.e.*, `Location.Value` in Table 2 (calculated from Table 3) is [L1: 0.54, L2: 0.35, L3: 0.11] and, thus, the determinized value of `location` in Table 1 is `L1`. Hence, the combined probability of t_1 satisfying all the selection conditions of the query is 0.54. The expected precision of the answer is 0.54. 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 of $\langle t_i, \mathcal{A}_j, f_k\rangle$, JENNER estimates 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 of the execution of f_k over the state of $\mathcal{A}_j$ in tuple t_i in the current database D_{w-1} be Δ. Such an uncertainty reduction measure Δ

allows 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. The tuple's entropy after the execution of f_k is $\mathcal{E}_{w-1} - \Delta$. We estimate the new probability p of $t_i.\mathcal{A}_j$ satisfying the selection condition by solving the following equation:

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

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 (denoted as p_{low}) and another that corresponds to the increase in probability (denoted as p_{high}). While JENNER does not depend on the probability to monotonically increase with the execution of more enrichment functions on each tuple, for the approach to be effective we expect that such would be the case for much of the data. The works on ensemble classifiers [23, 52] provide evidence that multiple classifiers together can reduce the inference error more as compared to the individual classifiers. For a single object, the enrichment functions can make mistakes in the prediction but overall, executing more enrichment functions increases the quality of the inference. Furthermore, several authors addressed the problem of reducing the cost of ensemble classifiers on large datasets. Authors either used a separate dataset to score the performance of classifiers and chose ensemble dynamically at inference time (in Dynamic Ensemble Selection mechanisms [18, 19]) or dynamically pruned away classifiers in an existing ensemble with accuracy lower than a threshold (in Ensemble Pruning mechanisms [28, 41]).

Example 3.2. Consider t_3 in Table 3 the value of which, based on the execution of the first two enrichment functions, is a distribution [0.15, 0.35, . . ., 0.05] over the possible locations. Given the query in Code Listing 1, the probability of t_3 satisfying the predicate on location is 0.15 and not satisfying is 0.85. As a result, entropy is calculated as 0.60. Consider the execution of third enrichment function and the associated entropy reduction as 0.3. With the new entropy of (0.6-0.3) =0.3, JENNER solves Equation 1 to determine p_{low} and p_{high} as 0.05 and 0.95 respectively. ■

Given (p_{low} and p_{high}) of the tuple, JENNER computes the probability of the tuple to be part of the real answer of the query in epoch e_w, denoted as $\mathcal{P}_{low}$ and $\mathcal{P}_{high}$ respectively (as described in Definition 2.2). Considering $\mathcal{P}_{low}$ and $\mathcal{P}_{high}$ of the tuple and the probabilities of other tuples satisfying the query condition (which is the same as in the previous epoch e_{w-1}), JENNER determines 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 $\mathcal{P}_{low}$ or it is $\mathcal{P}_{high}$. 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 $\widehat{Qty}(Ans_{high})$.

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

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 be negative depending upon the value of p_{w-1}. In this case, we consider the benefit to be 0 and such a function would not be chosen for enrichment.

3.2.2 *Generalizing to Other Queries* .

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

For queries with join conditions, the benefit is computed for each tuple of the relations separately, *i.e.*, JENNER does not compute the benefit of the composite tuples generated from two tuples of the different relations. This allows JENNER to measure the benefit of the tuples in linear time irrespective of the type of the query. 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}, [6] 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.*, at-least one tuple in the answer set of e_{w-1} was generated by these tuples) or not in the answer set (*i.e.*, no tuples in the answer set of e_{w-1} was generated by these tuples).

To determine R_i^{σ}, JENNER first determines Ans_{w-1} and finds out the tuple of R_i with minimum *match_prob* (*i.e.*, probability of satisfying all the selection conditions of R_i) that still qualified to be part of Ans_{w-1}. This minimum *match_prob* is denoted as the *relation-threshold* of R_i. The tuples with *match_prob* higher than this threshold forms R_i^{σ} and the remaining tuples form the set of $R_i^{\neg\sigma}$. 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 Ans^{real}, let us consider the projection of t_i to its constituent tuples in relations of $R_1, \ldots, R_n$. 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 is computed as follows: let the determinized value of $t_i[R_j]$ be $Det(t_i[R_j])$. The probability of $t_i[R_j]$ satisfying the join condition on 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 $\mathcal{A}_m$ 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 satisfying the join condition of $R_j.\mathcal{A}_m = R_k.\mathcal{A}_m$ is the product of the above two probabilities, *i.e.*, $Val(state(t_i[R_j].\mathcal{A}_m))[Det(t_i[R_j].\mathcal{A}_m)] \times Val(state(t_i[R_k].\mathcal{A}_m))[Det(t_i[R_k].\mathcal{A}_m)]$. 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 tuples $t_i \in Ans_w$ to be part of real answer set Ans^{real}, JENNER computes F_α measure by Equation 5. To compute benefit of a triple $\langle t_i, \mathcal{A}_j, f_k \rangle$, where t_i is in relation R_p and it is part of $R_p^{\neg\sigma}$, JENNER estimates the quality of the answer that would result if Q were to be executed on the database after executing

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

f_k on $t_i.\mathcal{A}_j$. JENNER follows the same strategy as selection queries where it generates p_{low} and p_{high} for the condition on $t_i.\mathcal{A}_j$ to be met. In each case, it re-executes the query, generates the answers Ans_{high} and Ans_{low}. As in selection queries, it chooses an answer with maximum quality (as described in §3.5).

The above way of computing benefit for executing f_k on $t_i.\mathcal{A}_j$ requires (*i*) determining the probabilities p_{low} and p_{high} from the entropy reduction of f_k, (*ii*) re-executing Q on the database state resulting in $D_{e_{w-1}}$ with the probability of $t_i.\mathcal{A}_j$ matching the query condition modified to p_{low} (or p_{high}), and (*iii*) run *ProduceQueryResult* in both cases (the complexity, as will be clear in §3.5 is $|Ans|log(|Ans|)$, where $|Ans|$ is the size of the answer from which query result is selected). Thus, the complexity of above three steps is $O(costQ + |Ans_w|log(|Ans_w|))$, where $costQ$ is the time to execute Q, $|Ans_w|$ is the size of 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. JENNER uses a greedy approach to choose this enrichment plan for the epoch e_w. For the ⟨tuple, derived attribute, enrichment function⟩ triples in $CandidateSet^M$, it computes the benefit as described earlier. The triples in CandidateSet are sorted in decreasing order of their *benefit*/*cost* values. The enrichment plan is chosen from the sorted set starting from the triple with highest *benefit*/*cost* value. Note that choosing the triples based on *benefit*/*cost*, allowed JENNER to achieve two goals: (*i*) If a triple has a very high benefit value but the cost of enrichment function is also high then such triples are not executed in the beginning and (*ii*) triples with smaller *benefit* and cost can be enriched in the beginning in large numbers to achieve higher improvement of the answer quality.

3.4 Execution of Enrichment Plan

This step executes the ⟨tuple, derived attribute, enrichment function⟩ triples present in EP_w. While executing an enrichment function on a set of tuples, JENNER batches the tuples together and then execute the enrichment function on them. 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. In an epoch, JENNER keeps executing the triples until the epoch duration is exhausted.

3.5 Produce Query Result

After the state update of tuples enriched in e_w, the original query Q is re-executed on the determinized representation to find the set of potential answer. For each tuple t_i in computed Ans_w, JENNER determines the probability of t_i to be in Ans^{real}, based on the probability of the tuples in base relations that constructed t_i. Instead of returning Ans_w, it returns a subset of tuples that maximizes the answer quality (*i.e.*, F_α-measure for set-based queries). For aggregation queries, JENNER first determines the set of answers that optimizes F_α-measure and then computes the aggregation function. Note that it is possible that a tuple that was returned as an answer in one of the previous epochs is retracted in the current epoch e_w.

JENNER is based on the following observation (proved in a theorem in [4]) : Let $t_1, t_2, \ldots, t_n$ be the set of tuples in Ans_w sorted by 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.

JENNER utilizes this observation where it sorts the tuples of Ans_w based on their probability of being in Ans^{real} and continue including answer tuples until $E(F_\alpha)$ measure of the answer is maximized. We refer to the probability of the last tuple that is part of the answer of epoch e_w as the *answer-threshold*. The time complexity of this step is $O(nlog(n))$ where n is the size of $CandidateSet^M$.

Example 3.3. Consider the query of Figure 1a with conditions on relations R_1 and R_2. Suppose the probe query results of R_1 and R_2 contained five tuples: $\langle r_1^1, r_2^1, \ldots, r_5^1 \rangle$ and ten tuples: $\langle r_1^2, \ldots, r_{10}^2 \rangle$ respectively. Without loss of generality, suppose the tuples of each relation are sorted by their probability of satisfying all the selection conditions on 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 shown in Example 3.4. JENNER keeps including the tuples in Ans_w until the $E(F_1)$ measure of the answer keeps increasing. Ans_w chosen in this way has maximum $E(F_\alpha)$ measure.

We next discuss how the probability of a tuple $t_i \in Ans_w$ to be in the true answer is calculated based on the corresponding tuples in base relations $R_i \in Q$ that formed t_i. For selection queries over a single derived attribute, it is the probability of the value of the tuple to match the 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 independence assumption of derived attributes. For queries where answer tuple t_i is formed using tuples from multiple relations, the corresponding probability is based on the product of probabilities of individual tuples to satisfy the individual selection and join conditions. [7] An example is shown below.

Example 3.4. Consider the query of Figure 1a on 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, $r_1.\mathcal{A}_1$ is a_1 (*i.e.*, the value with highest probability after determinization) and the probability associated with the value is 0.9. Similarly, let $r_2.\mathcal{A}_6$ be a_6 and the probability associated with it is 0.8. Considering the join condition on derived attribute (*i.e.*, $R_1.\mathcal{A}_3 = R_2.\mathcal{A}_3$), suppose the attribute values of the tuples after determinization on $\mathcal{A}_3$ match and the

[7] For duplicates, probability values are added up as in probabilistic databases [20].

Table 4: Queries used in the experiments.

ID	Query
Q1	Trajectory of a person in a time interval.
Q2	Users who came in contact with a specific user in a time interval.
Q3	Average time spent by a user in different infrastructure types.
Q4	Tweets with positive sentiment and of a particular topic.
Q5	Tweet pairs with same sentiment value posted between an interval.
Q6	Tweets with positive sentiment posted from a particular state between two time intervals.
Q7	Number of tweets posted for each topic between two time intervals.

Table 5: Datasets and cost/quality tradeoff of functions.

Relation	Derived attrs.	Function	Cost (ms)	Quality
WiFi 10M tuples 9GB Size	location(304)	LOC_2	24.5	0.68
		LOC_4	46.4	0.75
		LOC_8	93.7	0.82
		LOC_16	186.4	0.91
TweetData 11M tuples 10.5GB Size	sentiment(3)	SVM	1.67	0.61
		KNN	2.81	0.72
		GNB	5.32	0.81
		MLP	6.26	0.89
	topic(40)	LDA	2.17	0.58
		LR	3.89	0.67
		KNN	5.48	0.75
		GNB	7.82	0.88

corresponding probabilities 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 is $\mathcal{P}(\langle r_1, r_2\rangle) = 0.9 \times 0.8 \times 0.95 \times 0.85 = 0.58$. We compute these probabilities 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 the conditions in the query.

After determining the Ans_w that optimizes the quality metric $E(F_\alpha)$, JENNER prunes tuples from the candidate set whose impact on quality improvement of the answer set in subsequent epochs is expected to be low. JENNER finds out the tuples of each relation that already contributed to Ans_w of the current epoch and removing them from $CandidateSet^M$. Enriching such tuples has low impact on improving the quality of the query result as proved in [4].

3.6 Optimizing Benefit Computation

The techniques for benefit computation for an enrichment function f_k on attribute $t_i.\mathcal{A}_j$, as described in §3.2.1 and §3.2.2, used simulated execution of f_k to assess the impact of execution on the overall quality of the query answers. Their overall time complexity is $O(n(costQ + |Ans_w| log(|Ans_w|)))$ where n is the size of $CandidateSet^M$. This section presents a strategy wherein *SelectEnrichmentPlan* can select the plan without explicitly calculating the benefit of the enrichment functions. We define a metric, *RelativeBenefit*, that allows JENNER to order the triples as if they were ordered based on their *benefit*/*cost* value. This metric is derived from the expression of expected F_α measure in Equation 5. In deriving this metric, we choose two triples from $CandidateSet^M$ and derive how much expected F_α measure improvement per unit cost can be brought by enriching them in the current epoch. Then, we derive the condition in which one of the triples has higher expected F_α measure improvement per unit cost than the other. It is observed that their ordering depends on $\mathcal{P}_i$, *i.e.*, probability of the tuple satisfying the query condition in epoch e_{w-1}, $(\mathcal{P}_i + \Delta\mathcal{P}_i)$, *i.e.*, the new probability of t_i if it is enriched in the current epoch of e_w, and the cost of the enrichment function. We explain the metric first for selection queries and then generalize it.

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.

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

where $\mathcal{P}_i$ is the probability of the tuple satisfying the query in the previous epoch of e_{w-1} and $(\mathcal{P}_i + \Delta\mathcal{P}_i)$ is the value of $\mathcal{P}_{high}$, *i.e.*, the new probability of t_i satisfying the query if it is enriched in the current epoch of e_w. The value of $\mathcal{P}_{high}$ is calculated as described in Definition 2.2 and in Section 3.2.1. JENNER only uses $\mathcal{P}_{high}$, instead of $\mathcal{P}_{low}$, as by using the latter the benefit of the triple is 0. Below, we show that ordering two triples based on *RelativeBenefit* metric ensures that they are ordered based on their *Benefit*/*Cost* values.

Example 3.5. Consider the query of Figure 1a on relations R_1 and R_2 and two tuples $r_1 \in R_1$ and $r_2 \in R_2$. Suppose, the probability of r_1 satisfying all the conditions on derived attributes (*i.e.*, $\mathcal{P}_1$) be 0.8 in epoch e_{w-1}. Similarly, let the probability of the tuple r_2 be 0.7 in e_{w-1}. If r_1 is enriched in epoch e_w, let the new probability values of satisfying the query be as follows: $\mathcal{P}_{high} = 0.9$ and $\mathcal{P}_{low} = 0.1$. The *RelativeBenefit* of r_1 is $(0.8 \times 0.9)/0.04 = 18$. Similarly, if r_2 is enriched, let the new probabilities of r_2 be $\mathcal{P}_{high} = 0.75$ and $\mathcal{P}_{low} = 0.25$. Hence, the *RelativeBenefit* of r_2 is $(0.75 \times 0.7)/0.03 = 17.5$. Comparing the *RelativeBenefit* of tuples, JENNER ranks r_1 before r_2. Now, we check if enriching tuple r_1 improves the expected F_α measure of the answer set per unit cost more as compared to r_2. Recall that expected F_α measure is calculated using Equation 5. Considering that all the tuples, except r_1 and r_2, that were in the answer set of epoch e_{w-1}, still remain part of the answer in e_w, the numerator of F_α measure is increased by $\mathcal{P}_{high}$ in e_w. The denominator increases by the value of $(1 + \Delta\mathcal{P}_1) = 1 + (\mathcal{P}_{high} - \mathcal{P}_1)$. Let the numerator of expected F_α measure in e_{w-1} be 30 and the denominator be 50, *i.e.*, $F_\alpha = (30/50) = 0.6$. Hence, the F_α measure of the answer due to the enrichment of r_1 is $(30+0.9)/(50+1+0.1) = 0.6046$. Similarly, the F_α measure due to r_2 is $(30+0.75)/(50+1+0.05) = 0.6024$. Hence, the *Benefit* per unit cost of r_1 (*i.e.*, $(0.046/0.04) = 1.15$) is higher than r_2 (*i.e.*, $(0.024/0.03) = 0.8$).

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:*

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

Proof Sketch. 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 $\Delta\mathcal{P}_i < \Delta\mathcal{P}_q$. In each of these orders, we determine the answer set by calculating the new threshold probability value. After determining the answer set, the quality of the answer set is calculated separately both when triple (t_i, $\mathcal{A}_j$, f_k) and triple (t_q, $\mathcal{A}_s$, f_v) are enriched. We measure their benefit using Equation 7. By simplifying the equation, it is observed that the first triple will have higher benefit/cost than the second one when the condition in Equation 11 is satisfied. We provide the complete proof in [4]. ■

Given the above theorem, JENNER computes *RelativeBenefit* of the triples and selects an enrichment plan (§3.3). This results in a

Table 6: Exp 1. Query time without progressiveness (in Mins).

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Time	31	44.5	40.6	22.1	67.1	39.2	45.1

time complexity of $O(n)$, where n is the size of $CandidateSet^M$ as compared to the approach of computing benefits explicitly.

More General Queries. To exploit the previous strategy in other queries, we need to estimate the number of tuples that would result 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:

$$RelativeBenefit(t_i, \mathcal{A}_j, f_k) = \lambda_{R_i} \cdot \left[\frac{\mathcal{P}_i(\mathcal{P}_i + \Delta\mathcal{P}_i)}{c_k}\right] \quad (12)$$

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

Datasets. We used the following datasets to evaluate JENNER: (*i*) WiFi data containing 10M WiFi connectivity events of user's mobile devices in a university campus (taken from the SmartBench benchmark[29]) and (*ii*) TweetData containing 11 million tweets.

Enrichment Functions. The enrichment functions are presented in Table 5. For the wifi dataset, we used multiple versions of the algorithm in [38] (`LOC_n`). `LOC_n` analyzes the pattern of the locations visited by the user and the interaction between the user and other users to infer their location. It is implemented as a multi-tuple-input enrichment function with the tuples collected in the past n days as input. For the tweet dataset, we used the following probabilistic classifiers (single-tuple-input enrichment functions): Support Vector Machine (SVM), k-Nearest Neighbor (KNN), Gaussian Naïve Bayes (GNB), Multi-Layered perceptron (MLP), Linear Discriminant Analysis (LDA), and Logistic Regression (LR).

Queries. Table 4 shows the queries used in our experimental study. ***Q1-Q3*** are from the SmartBench benchmark [29] on the `wifi` dataset. Q1 is a simpler query with selection conditions on derived attributes while Q2 and Q3 require join on derived attributes. Q4-Q7 are analytical queries over Tweet data with different complexities - Q4 is a selection query, Q5-Q6 are join queries, and Q7 is an aggregation query. The SQL implementations of all queries are available in [4]. The epoch sizes in Experiment 1 are set according to the optimal epoch sizes determined by Experiment 5 for different queries. The quality of the query results are measured using the ground truth data available for the datasets. JENNER uses expected F_α measure and *RelativeBenefit* to select the enrichment plans without using ground truths. Note that, we do not consider the enrichment functions to be 100% accurate (i.e., executing all enrichment functions might result in an F_1 measure lower than 1).

Enrichment Plan Generation Strategies. We compare JENNER with three different 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 function with the highest value is executed on all tuples of the probe query result before choosing the next function. In an epoch, only tuples are chosen and they are enriched using the chosen function. (*iii*) *Sample-based with Random Order* (**RO**): that selects both tuples (from probe query results) and enrichment functions randomly.

Table 7: Exp 2. Total time of query execution and enrichment with varying selectivity of fixed condition in Q4.

Selectivity	TTR (90%)	TTR (95%)	Query Completion
100%	18.37 mins	25.19 mins	10 hours (timeout)
10%	5.88 mins	8.71 mins	4.48 hours
1%	25.19 sec	2.1 mins	27.29 mins

Table 8: Exp 3. Progressive Scores.

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	**0.74**	0.37	0.33	0.34
Q4	**0.80**	0.34	0.33	0.31					

4.1 Experimental Results

The experiments were performed on an enrichment server with 16 core 2.50GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD. The datasets were stored in two tables of a PostgreSQL database. If we had to perform complete enrichment of 11M tweets of TweetData table for both `topic` and `sentiment` attributes using the functions of Table 5, it would have taken ≈ **43 hours** to complete after using all the 16 cores of the server.[8] The authors in [21] also showed that sentiment inference on tweets using complex ML algorithms can take hours for the Sentiment140 dataset [27] with 1.6 million tweets. The complete enrichment of 100K wifi data for the derived attribute of `location` would have taken ≈ 37 days on the same server. Hence, complete enrichment of the data is infeasible.

Experiment 1 (Need for a Progressive Approach). We compare JENNER with the approach of completely enriching the objects required to answer the query first and then evaluating the query. The query execution times in the second approach are presented in Table 6. Since, in the second approach, all the required objects (*i.e.*, the objects present in the probe query result) are enriched first, all the queries have very high execution times. In contrast, if we execute the same queries in JENNER, users do not have to wait for a long time to receive the query results. Furthermore, the variation of the quality of the results with respect to time are presented in Figure 2. It shows that in JENNER, the quality of the query result achieves very high value within a few seconds of query execution.

Experiment 2 (Eager Enrichment VS. JENNER). We vary query selectivities to compare JENNER against the strategy of complete enrichment (*i.e.*, Eager). We define the selectivity as the ratio of output-cardinality to the input-cardinality of a query. We use Q4 and replace the predicate on `TweetTime` to control the selectivity. Table 7 shows the time to reach the qualities of 90%, 95%, and 100%

[8]We experimentally measured the runtime of the enrichment functions in the tweet dataset for 1 million tweets to be 3 hours 55 minutes. This runtime multiplied by a factor of 11, as the dataset has 11 million tweets, is ≈ 43 hours.

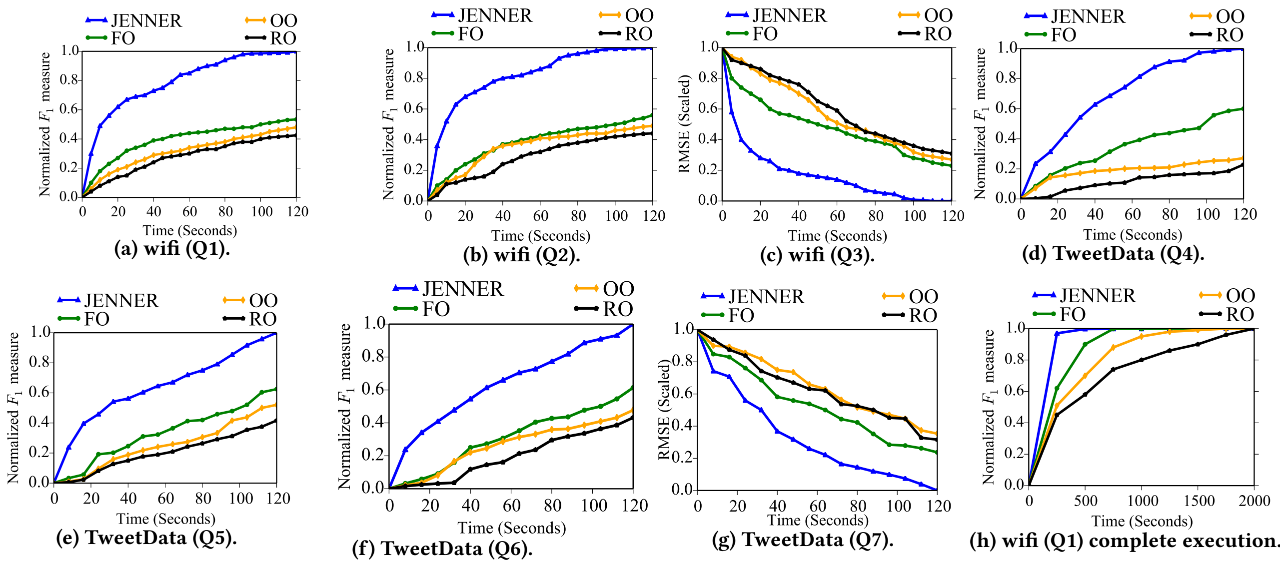

Figure 2: Exp 1 and Exp 3. Performance results of different plan generation strategies.

of the maximum quality in JENNER. Comparing Table 7 and the time required in the eager enrichment strategy (*i.e.*, ≈ 43 hours), we observe that the total time for query processing and enrichment in JENNER is much lower than the eager strategy. Furthermore, Table 7 shows that JENNER achieves 90% and 95% of the maximum quality within a few minutes of query execution. Only in the unfavorable case of 100% selectivity and 100% quality, the execution time is as high as the eager strategy. Even in such situations (not our design target) JENNER does not result in much overhead (as in Table 9), whereas it saves several orders of magnitude in the favorable situations of lower selectivity and quality requirement.

Experiment 3 (Comparison with Different Progressive Approaches). This experiment compares JENNER with FO, OO, and RO approaches. 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 2 where we measure the quality of the query result using *normalized* F_1 *measure* for set based queries and *normalized root-mean-square-error* (RMSE) for aggregation queries (Q3 and Q7). The normalized F_1 measure is calculated as F_1/F_1^{max}, where F_1^{max} is the maximum F_1 measure that is achievable by executing all the enrichment functions.[9] Similarly, normalized root-mean-square-error is calculated by measuring $RMSE/RMSE^{min}$ where $RMSE^{min}$ is the minimum RMSE achievable by executing all enrichment functions. Furthermore, we report the progressive scores in Table 8. Note that in Figures 2a-2g, the plots are capped to only the first two minutes of query execution as the time to enrich completely is large. When the queries are run for the duration of complete enrichment, all the approaches converge to the value of 1 for set-based queries (as shown for Q1 in Figure 2h) and the value of 0 for the aggregation query of Q7.

Figure 2 and Table 8 show that JENNER outperforms the other approaches significantly for all queries. With JENNER, the answer achieves a high quality within first few epochs of query execution. For example, Figure 2(a) shows that JENNER achieves F_1-measure of 0.9 within first 80 seconds. Furthermore, JENNER achieves a high rate of quality improvement in early epochs resulting in the highest progressive score among all the approaches. This is due to JENNER's strategy of monitoring the progress of enrichment in each epoch and dynamically selecting an enrichment plan.

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

Table 9: Exp 4. Overhead as compared to the total query execution time. The remaining time is used in enrichment.

Q	Plan Gen.	DBMS Time	Network Cost	Q	Plan Gen.	DBMS Time	Network Cost
Q1	0.64 %	0.37%	0.86%	Q5	1.32%	1.84%	2.90%
Q2	0.93%	0.52%	0.60%	Q6	0.71%	1.20%	2.71%
Q3	0.96%	0.73%	0.65%	Q7	1.33%	1.10%	1.40%
Q4	1.45%	0.70%	2.80%				

Experiment 4 (Overhead). Table 9 shows different *time* overheads of JENNER: (*i*) time spent in the enrichment plan generation, (*ii*) execution time in the DBMS, and (*iii*) the network cost of transmitting data between DBMS and enrichment server. We measured the total plan generation time across all epochs and reported the sum as the percentage of total time. For DBMS time, we measured the total cost of probe queries and the queries at the end of each epoch. For network cost, we measured the total cost of transmitting the result of probe queries at zero-th epoch and updating the states of tuples at the end of epochs. The results show that (*i*) the cost at DBMS increases when the query complexity increases (*e.g.*, see increase in DBMS time between Q4 and Q5); (*ii*) total overhead remains a small fraction of the overall execution time (ranging from 1.87% to 6.05%). Thus, JENNER 's adaptive approach of selecting tuples to enrich does not pose significant overhead. For storage overhead, the total size of *CandidateSet*, $CandidateSet^M$, and EP_w were less than 10 MB which is a small fraction of the data sizes (*i.e.*, 9 GB and 10.5 GB as shown in Table 5). Furthermore, the state table sizes for `wifi` and `TweetData` tables were 0.4 GB and 0.9 GB, respectively.

Experiment 5 (Epoch Size). This experiment studies how epoch size affects progressiveness achieved for the queries. Figure 3 plots: (*i*) Time to reach (TTR) 90% quality by JENNER for Q2 (1% selectivity) and (*ii*) Time overhead as percentage of total query execution time. Figure 3(a) shows that when the epoch size is reduced from 8 to 4 seconds, the TTR 90% reduces as JENNER chooses enrichment

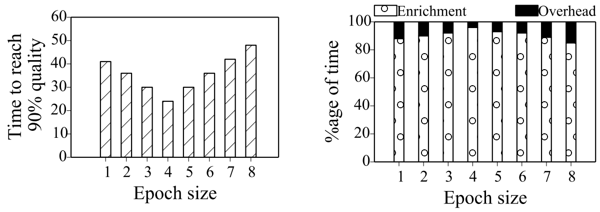

Figure 3: Exp 5. Effect of epoch sizes: (a) time to reach 90% of max. quality (Q2) and (b) %age of plan generation time.

Table 10: Exp 6. Avg. Number of Candidates in $CandidateSet^M$**.**

Query	JENNER	Naive	Query	JENNER	Naive
Q1	800	3000	Q5	11000	20000
Q2	1200	5000	Q6	6000	10000
Q3	16000	50000	Q7	500	1000
Q4	1200	2000			

Table 11: Exp 7. Percentage of enrichment plan generation time taken by using relative benefit as compared to benefit.

Query	rel. benefit	benefit	Query	rel. benefit	benefit
Q1	0.64%	32.17%	Q5	1.32%	94.17%
Q2	0.93%	61.45%	Q6	0.85%	58.96%
Q3	0.96%	82.38%	Q7	0.62%	43.14%
Q4	1.45%	88%			

plans more frequently. However, when it is reduced further from 4 to 2 seconds, the overheads due to frequent plan generation reduces the effective time for enrichment, thereby increases the TTR.

Experiment 6 (Impact of Pruning). As shown in §3.2, JENNER restricts tuples in the enrichment plan to only those that are not in the answer of previous epoch. We compare this strategy with the strategy of using all tuples in $CandidateSet(R_i)$ for generating enrichment plans. Table 10 shows that the size of $CandidateSet^M$ in JENNER is significantly smaller, hence reducing the cost of plan generation. The entries in $CandidateSet^M$ 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 7 (Impact of Optimized Benefit Estimation). This experiment compares JENNER when it uses the naive strategy (described in §3.2) of benefit estimation as compared to the strategy in §3.6 used in JENNER (with complexity $O(n)$). Table 11 presents the percentage of total execution time taken by the two approaches. The table shows that the naive strategy would have taken 32% to 94 % of total execution time to estimate benefit, thereby making JENNER impractical. Hence, benefit estimation using *RelativeBenefit* allowed JENNER to allocate most of the execution time to enrichment.

Experiment 8 (Accuracy of Different Estimation Steps). In each epoch e_w, for a tuple $t_i \in R_j$ that was in the probe query result, JENNER estimates the probability that it will be in the answer of e_w (*i.e.*, generate at-least one answer tuple). Furthermore, JENNER estimates the cardinality, *i.e.*, the number of answer tuples that will be generated by t_i. Both estimations are based on the answer returned in the previous epoch e_{w-1}. We measure accuracy of both estimations by calculating the Standard Deviation (SD) from the actual value (determined by ground truth) and the estimated value.

In an epoch, the deviation in estimated probability and cardinality is calculated for each tuple of the probe query result and then

Table 12: Exp 8. Accuracy of (a) probability estimation and (b) cardinality estimation.

Q	Std. Dev.	Q	Std. Dev.
Q1	1.18%	Q5	2.31%
Q2	1.87%	Q6	1.94%
Q3	2.03%	Q7	2.43%
Q4	2.11%		

Q	Std. Dev.
Q1	2.06%
Q2	2.37%
Q5	3.14%
Q6	2.74%

SD is computed over all such tuples. This process is continued over all epochs and SD is reported in Table 12. We report the accuracy of probability estimation in Table 12 (a) and cardinality estimation (for join queries) in Table 12 (b). These results highlight that JENNER provides accurate estimation for both the metrics.

5 RELATED WORKS

JENNER is related to several lines of prior research work. Progressive query answering was explored in approximate processing of aggregate queries [30]. The techniques offer error bounds based on sampling [6, 48] that improves when larger samples are used. However, they do not consider the problem of enrichment during query processing and cannot be used in our setting. Progressive data processing has also been considered in data cleaning contexts such as in entity resolution [9, 10, 42, 47]. JENNER adapted the metric for progressive enrichment from these works. Since JENNER explores progressive enrichment during query processing, it differs from these works as they did not consider progressive data cleaning in the context of queries. As mentioned in §1, data cleaning during query processing has been studied in [51]. However, such works did not consider progressive data processing. Recently, in [5] we explored a complementary challenge of supporting enrichment during query processing: using a loose design (enrichment inside a middleware) and a tight design (enrichment in DBMS). However, [5] does not consider the problem of ordering enrichment functions to optimize enrichment during progressive query processing as addressed in this paper. JENNER is also related to expensive predicate optimization of [35, 36, 40] which focused on predicate reordering during query processing to minimize cost. In contrast, JENNER focused on progressive enrichment during query processing.

6 CONCLUSIONS

We described an approach 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 trade-off 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 experiments validate the improvement achieved by JENNER over naive strategies to support progressiveness in query processing.

ACKNOWLEDGMENTS

This material was partially funded by the research sponsored by DARPA under agreement number FA8750-16-2-0021 and NSF Grants No. 1952247, 2133391, 2032525, and 2008993.

REFERENCES

[1] 2014. Internet Live Stats. http://www.internetlivestats.com. Accessed: 2022-06-30.

[2] 2022. A Case for Enrichment in Data Management Systems. https://github.com/dhrubajg246/papers/blob/main/ACaseForEnrichment.pdf. *ACM SIGMOD Record* (2022). Accessed: 2022-06-30.

[3] 2022. EnrichDB system. https://github.com/DB-repo/enrichdb.. Accessed: 2022-06-30.

[4] 2022. Longer version of paper and codebase. https://github.com/jennerrepo/jenner. Accessed: 2022-06-30.

[5] 2022. Supporting Complex Query Time Enrichment For Analytics. https://github.com/dhrubajg246/papers/blob/main/SupportingEnrichmentForAnalytics.pdf. (2022).

[6] Sameer Agarwal et al. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data *(EuroSys)*.

[7] Sanjay Agrawal et al. 2008. Scalable Ad-hoc Entity Extraction from Text Collections. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 945–957. https://doi.org/10.14778/1453856.1453958

[8] Abdulrahman Alsaudi, Yasser Altowim, Sharad Mehrotra, and Yaming Yu. 2021. TQEL: Framework for Query-Driven Linking of Top-K Entities in Social Media Blogs. *Proc. VLDB Endow.* 14, 11 (2021), 2642–2654.

[9] Yasser Altowim et al. 2018. ProgressER: Adaptive Progressive Approach to Relational Entity Resolution. *ACM Trans. Knowl. Discov. Data* 12, 3 (2018), 33:1–33:45.

[10] Yasser Altowim and Sharad Mehrotra. 2017. Parallel Progressive Approach to Entity Resolution Using MapReduce. In *ICDE*. IEEE Computer Society, 909–920.

[11] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V. Kalashnikov. 2015. QuERy: A Framework for Integrating Entity Resolution with Query Processing. *Proc. VLDB Endow.* (2015).

[12] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.

[13] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25–40.

[14] Sebastian Braun et al. 2021. Towards Efficient Models for Real-Time Deep Noise Suppression. In *ICASSP*. IEEE, 656–660.

[15] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. 2017. Query Optimization for Dynamic Imputation. *Proc. VLDB Endow.* 10, 11 (2017), 1310–1321. https://doi.org/10.14778/3137628.3137641

[16] Zhaoqi Chen et al. 2009. Exploiting context analysis for combining multiple entity resolution systems. In *SIGMOD Conference*. ACM, 207–218.

[17] Tyson Condie et al. 2010. Online aggregation and continuous query support in MapReduce. In *SIGMOD Conference*. ACM, 1115–1118.

[18] Rafael M.O. Cruz et al. 2015. META-DES: A dynamic ensemble selection framework using meta-learning. *Pattern Recognition* (2015).

[19] Rafael M.O. Cruz, Robert Sabourin, and George D.C. Cavalcanti. 2017. META-DES.Oracle: Meta-learning and feature selection for dynamic ensemble selection. *Information Fusion* (2017).

[20] Nilesh Dalvi and Dan Suciu. 2007. Efficient Query Evaluation on Probabilistic Databases. *The VLDB Journal* (2007).

[21] Nhan Cach Dang, María N. Moreno-García, and Fernando De la Prieta. 2020. Sentiment Analysis Based on Deep Learning: A Comparative Study. *Electronics* 9, 3 (2020). https://doi.org/10.3390/electronics9030483

[22] Thomas G. Dietterich. 1995. Overfitting and Undercomputing in Machine Learning. *ACM Comput. Surv.* 27, 3 (1995), 326–327. https://doi.org/10.1145/212094.212114

[23] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma. 2020. A survey on ensemble learning. *Frontiers of Computer Science* 14, 2 (2020), 241–258.

[24] Alex Galakatos, Andrew Crotty, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2017. Revisiting Reuse for Approximate Query Processing. *Proc. VLDB Endow.* 10, 10 (2017), 1142–1153.

[25] Sahaj Garg, Anirudh Jain, Joe Lou, and Mitchell A. Nahmias. 2021. Confounding Tradeoffs for Neural Network Quantization. *CoRR* abs/2102.06366 (2021).

[26] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *SIGMOD Conference*. ACM, 805–815.

[27] Alec Go et al. 2009. Twitter sentiment classification using distant supervision. http://help.sentiment140.com/home. , 2009 pages.

[28] Huaping Guo, Weimei Zhi, Xiao Han, and Ming Fan. 2011. A New Metric for Greedy Ensemble Pruning. In *Artificial Intelligence and Computational Intelligence*, Hepu Deng, Duoqian Miao, Jingsheng Lei, and Fu Lee Wang (Eds.).

[29] Peeyush Gupta, Michael J. Carey, Sharad Mehrotra, and Roberto Yus. 2020. SmartBench: A Benchmark For Data Management In Smart Spaces. *Proc. VLDB Endow.* 13, 11 (2020), 1807–1820. http://www.vldb.org/pvldb/vol13/p1807-gupta.pdf

[30] Joseph M. Hellerstein et al. 1997. Online Aggregation. *SIGMOD* (June 1997), 171–182. https://doi.org/10.1145/253262.253291

[31] Alex Holub, Pietro Perona, and Michael C. Burl. 2008. Entropy-based active learning for object recognition. In *CVPR Workshops*. IEEE Computer Society, 1–8.

[32] Rein-Lien Hsu et al. 2002. Face detection in color images. *IEEE Tran. on Pattern Analysis and Machine Intelligence* (2002). https://doi.org/10.1109/34.1000242

[33] Rob J. Hyndman and Anne B. Koehler. 2006. Another look at measures of forecast accuracy. *International Journal of Forecasting* 22, 4 (2006), 679 – 688. https://doi.org/10.1016/j.ijforecast.2006.03.001

[34] P JACCARD. 1901. Etude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull Soc Vaudoise Sci Nat* 37 (1901), 547–579. https://ci.nii.ac.jp/naid/10019961020/en/

[35] Manas Joglekar et al. 2015. Exploiting Correlations for Expensive Predicate Evaluation *(SIGMOD)*. ACM, New York, NY, USA.

[36] Iosif Lazaridis et al. 2007. Optimization of Multi-version Expensive Predicates *(SIGMOD)*.

[37] Jian Li and Amol Deshpande. 2009. Consensus answers for queries over probabilistic databases. In *PODS*. ACM, 259–268.

[38] Yiming Lin et al. 2020. LOCATER: Cleaning WiFi Connectivity Datasets for Semantic Localization. *Proc. VLDB Endow.* 14, 3 (2020), 329–341. https://doi.org/10.5555/3430915.3442432

[39] Yiming Lin et al. 2021. T-cove: an exposure tracing system based on cleaning wi-fi events on organizational premises. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2783–2786.

[40] Yao Lu et al. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates *(SIGMOD)*. ACM, New York, NY, USA.

[41] Zhenyu Lu, Xindong Wu, Xingquan Zhu, and Josh Bongard. 2010. Ensemble Pruning via Individual Contribution Ordering *(KDD)*.

[42] D. Marmaros et al. 2013. Pay-As-You-Go Entity Resolution. *IEEE TKDE* (2013).

[43] Krystian Mikolajczyk et al. 2004. Human Detection Based on a Probabilistic Assembly of Robust Part Detectors. In *ECCV*.

[44] Barzan Mozafari. 2017. Approximate Query Engines: Commercial Challenges and Research Opportunities. In *SIGMOD Conference*. ACM, 521–524.

[45] Rabia Nuray-Turan, Dmitri V. Kalashnikov, Sharad Mehrotra, and Yaming Yu. 2012. Attribute and object selection queries on objects with probabilistic attributes. *ACM Trans. Database Syst.* 37, 1 (2012), 3:1–3:41.

[46] R. Olfati-Saber et al. 2005. Consensus Filters for Sensor Networks and Distributed Sensor Fusion *(CDC '05)*.

[47] T. Papenbrock et al. 2015. Progressive Duplicate Detection. *IEEE TKDE* (2015).

[48] Yongjoo Park et al. 2018. VerdictDB: Universalizing Approximate Query Processing *(SIGMOD)*.

[49] John C. Platt. 1999. Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*.

[50] David Powers et al. 2011. Evaluation: From precision, recall and F-measure to ROC, informedness, markedness & correlation. *J. Mach. Learn. Technol* 2 (01 2011), 2229–3981. https://doi.org/10.9735/2229-3981

[51] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System *(VLDB)*.

[52] Ye Ren, Le Zhang, and Ponnuthurai N Suganthan. 2016. Ensemble classification and regression-recent developments, applications and future directions. *IEEE Computational intelligence magazine* 11, 1 (2016), 41–53.

[53] Christopher De Sa, Megan Leszczynski, Jian Zhang, Alana Marzoev, Christopher R. Aberger, Kunle Olukotun, and Christopher Ré. 2018. High-Accuracy Low-Precision Training. *CoRR* abs/1803.03383 (2018).

[54] Nadia Felix F. Da Silva et al. 2016. A Survey and Comparative Study of Tweet Sentiment Analysis via Semi-Supervised Learning. *ACM Comput. Surv.* (2016).

[55] S. Singh et al. 2007. Indexing Uncertain Categorical Data. In *ICDE*. 616–625.

[56] Panos Vassiliadis. 2009. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)* 5, 3 (2009), 1–27.

[57] Xin Wang et al. 2018. SkipNet: Learning Dynamic Routing in Convolutional Networks. In *ECCV (13) (Lecture Notes in Computer Science, Vol. 11217)*. Springer, 420–436.

[58] Cort J Willmott and Kenji Matsuura. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate research* 30, 1 (2005), 79–82.

[59] Jie Xu, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2015. Query Aware Determinization of Uncertain Objects. *IEEE Trans. Knowl. Data Eng.* 27, 1 (2015), 207–221.

[60] Brandon Yang et al. 2019. CondConv: Conditionally Parameterized Convolutions for Efficient Inference. In *NeurIPS*. 1305–1316.

[61] Bianca Zadrozny and Charles Elkan. 2002. Transforming Classifier Scores into Accurate Multiclass Probability Estimates *(KDD)*.

[62] Matei Zaharia et al. 2012. Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing.

[63] Danyang Zhu et al. 2020. Efficient Precision-Adjustable Architecture for Softmax Function in Deep Learning. *IEEE Trans. Circuits Syst.* 67-II, 12 (2020), 3382–3386.