A Declarative Query Processing System for Nowcasting

Dolan Antenucci
University of Michigan
Ann Arbor, MI
dol@umich.edu

Michael R. Anderson
University of Michigan
Ann Arbor, MI
mrander@umich.edu

Michael Cafarella
University of Michigan
Ann Arbor, MI
michjc@umich.edu

ABSTRACT

Nowcasting is the practice of using social media data to quantify ongoing real-world phenomena. It has been used by researchers to measure flu activity, unemployment behavior, and more. However, the typical nowcasting workflow requires either slow and tedious manual searching of relevant social media messages or automated statistical approaches that are prone to spurious and low-quality results.

In this paper, we propose a method for declaratively specifying a nowcasting model; this method involves processing a user query over a very large social media database, which can take hours. Due to the human-in-the-loop nature of constructing nowcasting models, slow runtimes place an extreme burden on the user. Thus we also propose a novel set of query optimization techniques, which allow users to quickly construct nowcasting models over very large datasets. Further, we propose a novel query quality alarm that helps users estimate phenomena even when historical ground truth data is not available. These contributions allow us to build a *declarative nowcasting data management system*, RaccoonDB, which yields high-quality results in interactive time.

We evaluate RaccoonDB using 40 billion tweets collected over five years. We show that our automated system saves work over traditional manual approaches while improving result quality—57% more accurate in our user study—and that its query optimizations yield a 424x speedup, allowing it to process queries 123x faster than a 300-core Spark cluster, using only 10% of the computational resources.

1. INTRODUCTION

The past several years have seen a growing interest in social media *nowcasting*, which is the process of using trends extracted from social media to generate a time-varying signal that accurately describes and quantifies a real-world phenomenon. For example, the frequency of tweets mentioning unemployment-related topics can be used to generate a weekly signal that closely mirrors the US government's unemployment insurance claims data [10]. Researchers have applied nowcasting to flu activity [17], mortgage refinancings [12], and more [13, 14, 34, 37].

Although nowcasting has had more research attention than wide adoption, it has the potential for massive impact. Building datasets with nowcasting is likely to be faster and less expensive than traditional surveying methods. This can be a great benefit to many fields—economics, public health, and others—that to a computer scientist appear starved for data. For example, US government economists use a relatively small number of expensive conventional economic datasets to make decisions that impact *trillions of dollars* of annual economic activity. Even a tiny improvement in policymaking—enabled by nowcasting datasets—could mean the addition of billions of dollars to the economy.

Nowcasting research to date has been driven by research into particular *target phenomena*; the software has been ad hoc rather than general-purpose. Yet, the literature [12–14, 34, 37] and our first-hand experience [10, 11] suggests that building a nowcasting model entails a standard database interaction model, the human-in-the-loop workflow:

1. Select: The user selects one or more *topics* about the phenomenon derived from social media messages. A topic consists of a descriptive label (e.g., "job loss") and a time-varying signal (e.g., daily frequency of tweets about job loss).

2. Aggregate: Using an aggregate function (e.g., averaging), the user combines the selected topics into a single *nowcasting result*. The result consists of a set of topic labels (e.g., *job loss* and *searching for jobs*) and a single time-varying signal.

3. Evaluate: The user evaluates the nowcasting result. A good result contains relevant topic labels (e.g., they are about unemployment) and a high-quality signal (e.g., it closely tracks real-world unemployment).

The nowcasting user repeats the above loop until reaching success in Step 3 or giving up. Unfortunately, the user has the difficult task in Step 1 of trying to select topics that will optimize two potentially competing criteria. As a result, the traditional approach to nowcasting topic selection—hand-choosing topics with seemingly relevant descriptive labels—is tedious and error-prone. For example, in one of our user studies, participants employing this manual approach achieved 35% lower quality results than our automated solution and exerted much more effort in doing so (see Manual in Section 6.3). We illustrate this with the following example:

Example 1. *Imagine a government economist, Janet, who wants to use social media to estimate the target phenomenon of unemployment behavior. She transforms a database of social media messages into a database of social media topics.*

Proceedings of the VLDB Endowment, Vol. 10, No. 3

*She then **selects** all topics that contain the word "fired." She uses historical ground truth data to train a statistical model that **aggregates** the topics' signals into a nowcasting result. When **evaluating**, Janet finds the output signal poorly tracks unemployment. She also notes that many "fired" topics actually reflect news about war-torn countries in which missiles were "fired." She chooses different topics about "benefits," but the new result also tracks unemployment poorly; she sees that the messages describe "unemployment benefits" as well as the film, "Friends With Benefits." This loop continues dozens of times until she obtains a satisfactory result.*

A nowcasting software system might follow the naïve strategy of ranking all possible topic candidates by both signal and label relevance to Janet's target phenomenon. Such a system would still be burdensome to use. Consider:

EXAMPLE 2. *Janet visits the hypothetical nowcasting system, enters a query for "unemployment" and provides historical data for the US monthly unemployment rate. The system scores and ranks all topics by correlation with the ground truth signal and their labels' semantic relevance to "unemployment." The system then **selects** the top-scoring topics and **aggregates** them together into a nowcasting result. Janet has successfully avoided much of the manual trial-and-error process in Example 1, but due to the massive number of candidates, the new system takes a considerable amount of time—anywhere from fifteen minutes to several hours[1]—to compute results. After **evaluating** the result, she finds that many of the topics' social media messages do not reflect on-the-ground observations, but rather discuss the government's monthly data announcement. She modifies her query to use a weekly unemployment dataset instead of the monthly one and waits again for an extended time to find she has been only partially successful in avoiding the monthly unemployment announcement. She changes her text query from "unemployment" to "job loss, hired, benefits," waits yet again for an extended period, and finally obtains a good result.*

System Requirements — While using the feature selection system in Example 2 has allowed Janet to avoid many of the tedious iterations of Example 1's ad hoc nowcaster, some amount of iterating is unavoidable. Even with high-quality automatic feature selection, Janet will inevitably want to tweak the nowcaster to use different historical data or avoid certain social media topics, but these tweaks are now incredibly painful, as she waits for standard feature selection techniques to complete. In contrast, an ideal nowcasting system would combine high-quality feature selection with interactive-speed runtimes, so Janet can quickly make a small number of effective revisions. (Our target usage model is that of web search, where users can iteratively improve queries very rapidly, so even fifteen minutes can seem like an eternity when having to repeatedly refine ineffective queries.)

In this paper, we propose a framework for a *declarative nowcasting data management system*. The user expresses a desired target phenomenon, and in interactive time, the system automatically selects social media topics to satisfy the competing statistical and semantic relevance criteria. The user can now enjoy the benefits of automatic feature selection (fewer iterations) with the advantages of fast execution (quick iterations when strictly necessary). We are unaware of any existing data systems that can do this. Our initial prototype of this framework is called RACCOONDB.

[1]As seen with our baseline systems in Section 6.2.

Technical Challenge — Our automated method for predicting phenomena can be viewed as a form of multi-criteria feature selection from the set of all distinct social media topics—a truly massive set (e.g., we have 150 million topics in our experiments). Previous work in feature selection [40] has integrated selection with model construction, but existing approaches require several seconds to process a few hundred features—too slow for our intended application. Another area of related work is multi-criteria ranking [16, 21], but existing systems assume candidate scores are already provided or inexpensive to compute; in contrast, obtaining our scores is costly. Processing nowcasting queries requires a different approach that can choose the best topics from hundreds of millions of candidates in interactive time.

Our Approach — We expand upon the work of feature selection and multi-criteria ranking to handle the scale and speed required by nowcasting. We apply *candidate pruning* methods that exploit both signal and semantic information, along with several *low-level optimizations*, to achieve interactive-speed query processing. We also employ *query expansion* principles to find relevant results that user queries do not explicitly declare. Additionally, *user query log statistics* help identify when a query cannot be answered effectively.

Contributions — Our contributions are as follows:

- We define a novel query model for declaratively specifying a nowcasting model, which allows users to estimate real-world phenomena with little effort (Section 2).
- We propose a novel framework for generating nowcasting results that uses both semantic and signal information from social media topics to generate high-quality nowcasting results (Section 3).
- We describe a novel set of query optimizations that enable query processing in interactive time (Section 4).
- We propose a novel method for detecting low-quality nowcasting queries—even for targets lacking historical ground truth data—alerting users when results may be unreliable (Section 5).
- We built a prototype system, RACCOONDB, and evaluated it using 40 billion tweets collected over five years. We show that it is 57% more accurate than manual approaches in our user study and that its optimizations yield a 424x speedup. Compared to a 300-core Spark cluster, it processes queries on average 123x faster using only 10% of the computational resources (Section 6).

We note that this paper builds upon our past nowcasting work. Our early work in economics [10] was successful at nowcasting US unemployment behavior, but building that nowcasting model entailed the tedious process of manual searching. In later computer science work [11], we showed the potential of using topic label semantics to select relevant topics for nowcasting; however, that system had two crucial flaws: it did not use any signal information to find relevant topics—excluding an important dimension for determining topic relevancy—and it took hours to process a single query. In contrast, this paper's contributions enable finding more relevant topics and generating higher-quality results, all within interactive runtimes. In a short demonstration paper [9], we described the user interaction of RACCOONDB, which includes a web-based interface for users to easily create nowcasting queries and explore results in interactive time.

2. PROBLEM STATEMENT

We will now formally define the nowcasting data management problem. Table 1 summarizes our notation.

Nowcasting — The goal of a nowcasting system is to use social media data to produce a time-varying signal that describes a real-life, time-varying phenomenon. As the name implies, nowcasting systems estimate a *current* quantity based on *current* social media data. They do not attempt to predict future events or one-off events such as riots. Nowcasting is possible and reasonable in cases where a target phenomenon—such as unemployment, or going to the movies—also yields discussion in social media. For phenomena with little or no discussion, nowcasting will not be applicable.

Social Media Data — A nowcasting system operates on a *social media database* comprised of social media messages.

DEFINITION 1 (SOCIAL MEDIA DATABASE). *A social media database is a set $\mathcal{M}$, where each $m \in \mathcal{M}$ is a tuple $m = (msg, tstamp)$ containing user-created textual content (msg) and a timestamp of the content creation date ($tstamp$).*

From $\mathcal{M}$, a set of topics can be extracted into a *topic database* to represent different trends on social media.

DEFINITION 2 (TOPIC DATABASE). *A topic database is a set $\mathcal{T}$ where each topic $t \in \mathcal{T}$ is a tuple $t = (l, s)$ such that:*

1. *Topic t is associated with a set of messages $M_t \subset \mathcal{M}$.*
2. *Label l is a textual description of the messages in M_t.*
3. *Time-varying signal $s = \{s_1, \ldots, s_c\}$ is defined by operator $S : M_t \rightarrow s$, where $s_i = (time_i, y_i)$. Value y_i is the number of messages in M_t occurring in time period $time_i$, which is in a chosen domain (weekly, etc.).*

As an example, a topic with label "Christmas" may have a set of messages discussing the holiday and a signal s that spikes in December of each year. To extract topics from $\mathcal{M}$, several nowcasting projects have used simple string containment in the messages [10–12,14,17] (e.g., a single topic might be all the messages that contain, "I lost my job"). This approach has the advantage of supplying an easy-to-understand label for each topic. Others have used sentiment analysis [13] (e.g., a single topic might be all the messages that connote happiness). Similarly, methods like topic modeling [20] have shown success in topic extraction.

User Queries — In order to find relevant topics for a target phenomenon, users describe their target with a *user query*.

DEFINITION 3 (USER QUERY). *A user query is a tuple $\mathcal{Q} = (q, r)$ such that:*

1. *Query string q describes the target phenomenon.*
2. *Time-varying signal $r = \{r_1, \ldots, r_c\}$ describes the target's historic trend, where $r_i = (time_i, y_i)$. Value $time_i$ is in the same domain as $\mathcal{T}$, and y_i is a real number.*

Query string q is distinct from the topics above; a single q = "unemployment" could describe topics related to *losing a job*, *needing a job*, and so on. Users have two options for declaring query signal r. In the *standard query model*, r would be historical data of the target phenomenon, such as government survey or administrative data. For some phenomena, this ground truth data is not available; however, a popular practice in machine learning is to use synthetic or incomplete data instead of a traditional dataset; these *distantly supervised learners* are often able to achieve high quality results even when no good traditional data is available [28]. This motivated our *distant supervision query model*, where r would be a partial signal that describes the user's domain knowledge. For example, our economist knows that unemployment always spikes when Christmas seasonal jobs end, so she creates a signal r that shows spikes at the end of each year. Under this model, RACCOONDB processes queries as it does in the standard model, but it also provides a query quality alarm to help users identify low-quality queries (further discussed in Section 5).

Notation	Description
$\mathcal{M}$	Social media database
$\mathcal{T}$	Topic database
$\mathcal{Q} = (q, r)$	User query with query string q and signal r
$O = (A, o)$	Nowcasting result with label set A and signal o
$\mathcal{R}$	Topics in $\mathcal{T}$ used to create output O
k	Target size of $\mathcal{R}$
β	User preference for signal vs. semantic weighting

Table 1: Frequently used notations.

Nowcasting Results — For a given user query Q, RACCOONDB creates $\mathcal{R}$, a set of k topics selected from $\mathcal{T}$, which are then used to generate a *nowcasting result* that estimates the target phenomenon using social media.

DEFINITION 4 (NOWCASTING RESULT). *A nowcasting result is a tuple $\mathcal{O} = (A, o)$ such that:*

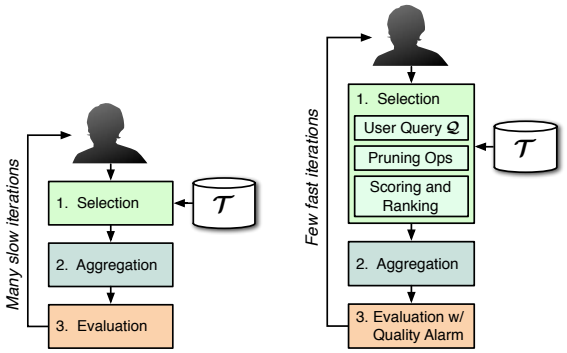
1. *Topic label set A is the set of topic labels in $\mathcal{R}$.*
2. *Output signal $o = \{o_1, \ldots, o_c\}$ is a time-varying signal defined by a given aggregate function $f : \mathcal{R} \rightarrow o$, where $o_i = (time_i, y_i)$. Value y_i is an estimate of the target in period $time_i$, which is in the same time domain as $\mathcal{T}$.*

Several choices for the aggregation function f are found in past nowcasting projects, such as simply combining the signals [12,17] or choosing the most important factor(s) from a principal components calculation [10,11]. The size of k will vary for different aggregation functions, but it is generally on the order of tens or hundreds.

Evaluating Success — The quality of nowcasting result $\mathcal{O}$ is measured using two criteria. First, *semantic relevance* measures the relatedness of output topic label set A to query string q. It is unacceptable to build a nowcasting result using a topic that is semantically unrelated to the researcher's goal, even if such a topic has been highly correlated with the target phenomenon in the past. For example, in certain time periods, the topic *pumpkin muffins* will likely yield a time series that is correlated with *flu activity*, because both spike as the weather cools. However, it is highly unlikely the two are causally linked: pumpkin muffin activity may fluctuate because of harvest success or changes in consumer tastes, whereas flu activity may fluctuate because of vaccine efficacy or prevalence of certain viruses.[2] We will evaluate output label set A using human annotators.

The second criterion is *signal correlation*, which measures the similarity between output signal o and query signal r. For both our query models, r represents (either directly or indirectly) the target phenomenon's behavior over time, and o should track this closely. We use Pearson correlation to measure this, a popular measure of signal similarity.

[2] Some seemingly unrelated topics may actually be quite indicative (e.g., maybe pumpkin muffins suppress the immune system), but this sort of *hypothesis generation* is different from our paper's focus, which is on *model generation*.



(a) Traditional **(b)** RaccoonDB

Figure 1: A comparison of the traditional approach to nowcasting and the RaccoonDB architecture.

In nowcasting practice, the relative weight of these two criteria—call them *semScore* and *sigScore*—is quite unclear, owing to the informal evaluation process for output label set A. RaccoonDB will allow the user to control the relative importance of these criteria with a parameter β ($\beta > 1.0$ favors semantic scoring, while $\beta < 1.0$ favors signal scoring), which we use to compute *harmonic mean scores* using a formula we borrow from an evaluation metric used in information retrieval, *generalized F-score* [33]. It is defined as:

$$F_{\boldsymbol{\beta}} = (1 + \beta^2) \cdot \frac{sigScore \cdot semScore}{(\beta^2 \cdot sigScore) + semScore} \quad (1)$$

We can now formally state the nowcasting data management problem we are solving with RaccoonDB:

Problem 1. *Given user query $\mathcal{Q}$ and weighting preference β, find a nowcasting result $\mathcal{O}$ that maximizes $F_{\boldsymbol{\beta}}$ over all choices of $\mathcal{R} \subset \mathcal{T}$.*

Further, because of the human-in-the-loop nature of building nowcasting models (Section 1), we want to find a result quickly enough to allow the user to interactively refine her query over many iterations (i.e., within several seconds).

3. SYSTEM ARCHITECTURE

In the traditional approach to nowcasting (Figure 1a), users spend many slow iterations manually searching for topics and evaluating their effectiveness (or manually filtering out spurious topics from statistical ranking approaches). In contrast, RaccoonDB introduces several novel components that allow for fewer, yet faster iterations and more expressive querying, as shown in Figure 1b.

First, RaccoonDB uses the two-part query $\mathcal{Q}$ to find topics in $\mathcal{T}$ that match well in both the semantic and signal domains (*Scoring and Ranking*). Additionally, *Pruning Optimizations* are used to avoid scoring and ranking irrelevant topics (further described in Section 4). RaccoonDB performs the same aggregation as the traditional workflow, and then, before showing the results to the user for evaluation, RaccoonDB uses a *Quality Alarm* to warn users of potentially low-quality queries (further discussed in Section 5).

3.1 Scoring and Ranking

The set of social-media-derived topics $\mathcal{T}$ contains a large number of topics (roughly 150 million in our experiments), which in principle need to be scored for signal and semantic relevance, then sorted, for *each novel nowcasting query*. Performing this step is the primary bottleneck in obtaining interactive query times in the naïve implementation of RaccoonDB (Algorithm 1), where similarity scores are generated in two parts for all $t \in \mathcal{T}$ and the user's query $\mathcal{Q}$.

Algorithm 1 Naïve Query Processing

Input: $\mathcal{T}$, $\mathcal{Q}$, k, β
1: $semScores$ = computeSemScores($\mathcal{T}$, $\mathcal{Q}$)
2: $sigScores$ = computeSigScores($\mathcal{T}$, $\mathcal{Q}$)
3: $\mathcal{R}$ = getTopFScores($semScores$, $sigScores$, β, k)
4: $factors$ = PCA($\mathcal{R}$)
5: (A, o) = factorCombiner($factors$, $\mathcal{Q}$)
6: $isLowQuality$ = queryQualityAlarm(A, o)
7: return (A, o, $isLowQuality$)

Algorithm 2 Thesaurus Semantic Scoring

Input: q, l
1: $toks1$ = tokenizeAndStem(q)
2: $toks2$ = tokenizeAndStem(l)
3: $bests1$ = [], $bests2$ = []
4: **for all** t1 in toks1 **do**
5: $scores1$ = []
6: **for all** t2 in toks2 **do**
7: $scores1$.append(jaccardDistance($t1$, $t2$))
8: **end for**
9: $bests1$.append(min($scores1$))
10: **end for**
11: **for all** t2 in toks2 **do**
12: $scores2$ = []
13: **for all** t1 in toks1 **do**
14: $scores2$.append(jaccardDistance($t1$, $t2$))
15: **end for**
16: $bests2$.append(min($scores2$))
17: **end for**
18: return 1 - (average($bests1$) + average($bests2$))/2

Semantic Scoring — On line 1 of Algorithm 1, RaccoonDB calculates a semantic similarity score between each topic's label and the user query string q. Several existing methods can be used for this, including pointwise mutual information [15] and thesaurus- or lexical-based metrics [26,35]. We chose a variation of thesaurus-based similarity for its simplicity and performance, as well as its shared spirit with query expansion principles, allowing topics that share no immediate tokens with a user query to still be relevant. Algorithm 2 describes how this is computed. On lines 1–2, the topic label l and the query string q are tokenized and stemmed with Porter stemming. Then on lines 4–18, for each pair of topic and query tokens, we compute the Jaccard distance between the tokens' thesaurus sets; all such distances are then combined using a method from Tsatsaronis, et al. [35], where an average is produced across them.

Signal Scoring — On line 2 of Algorithm 1, a signal similarity score is calculated between each topic's signal and the user's query signal r. Several existing methods can be used for this, including correlation-based metrics (Pearson, Spearman, etc.) and mean squared error. We use Pearson correlation due to its simplicity and popularity, as well as its statistical properties that we can exploit in our query processing optimizations (Section 4.1.1). If the user provides an incomplete query signal r (i.e., our distant supervision query model), we only compute Pearson correlation for the timespans explicitly provided.

Harmonic Mean Scoring and Ranking — On line 3 of Algorithm 1, RaccoonDB calculates a harmonic mean score (F_β) for each topic in $\mathcal{T}$ using Equation 1 and the user-provided weighting preference β. Finally, the topics are

sorted by their harmonic mean scores, and the top-k topics are selected as $\mathcal{R}$.

Time Complexity — For the naïve implementation of RaccoonDB (Algorithm 1), if topic signals have c observations and topic labels have on average u tokens, then for the n topics in $\mathcal{T}$, RaccoonDB scoring has $O((c+u)n)$ worst-case time. Since c and u are relatively small and slow-growing, this complexity can be simplified to $O(n)$. Ranking the topics requires $O(n\log(n))$ sorting, so the non-optimized system's complexity becomes $O(n+n\log(n))$, which can be simplified to $O(n\log(n))$ worst-case time. In Section 4.4, we discuss the time complexity of our optimized system, which drastically reduces the number of items scored and ranked.

3.2 Aggregation

A core part of nowcasting projects is aggregating the topics in $\mathcal{R}$ into the output signal o. Past systems have used a variety of techniques, including summing signals together [12, 17], dimensionality reduction [10, 11], or simply setting $k = 1$ (thus, no aggregation). This module is not a core contribution of RaccoonDB but is a pluggable component chosen by the domain experts using the system. By default, we use a form of principal component analysis (PCA) aggregation with $k = 100$, based on our past economic work [10].

We first perform dimensionality reduction on the topic signals in $\mathcal{R}$ using PCA (line 4 of Algorithm 1). We use this output to generate k factors that represent weighted combinations of the signals in $\mathcal{R}$. We then choose a subset of them using a heuristic based on a *scree plot* approach to choosing principal components. In traditional scree plot usage, a researcher plots each component's captured variance and eigenvalue and then looks for for an "elbow" in this plot, defining a break between *steep* and *not steep* portions of the plot. We mimic this process by looking for the first set of neighboring points where the delta between them drops below a fixed threshold that we tuned across a workload of queries. Finally, we use this subset as input to a linear regression task, in which the user's signal r is the training data. To prevent overfitting, the system always chooses a subset size so that it is smaller than the number of points in r. The output of this regression is the query output o. This aggregation has a negligible effect on the time complexity of RaccoonDB since the input of our aggregation has only k items and $k \ll |\mathcal{T}|$, the input size of our scoring and ranking procedures, which dominates the overall time complexity.

4. QUERY OPTIMIZATIONS

In order to process queries in interactive time, RaccoonDB automatically balances several optimizations according to the characteristics of the user query $\mathcal{Q} = (q, r)$ and weighting preference β. RaccoonDB uses two types of candidate pruning to avoid fully scoring all items in $\mathcal{T}$, the first using signal information and the second using semantics. The parameters of these pruning methods are adjusted for each individual query. Further, shared computation methods like SIMD instructions and parallelization, which are applicable to all user queries, allow RaccoonDB to further speed up candidate scoring. We discuss each of these below.

4.1 Candidate Pruning

We will now discuss how RaccoonDB performs confidence interval signal pruning and semantic pruning.

Algorithm 3 Confidence Interval Signal Pruning

Input: $\mathcal{T}$, $\mathcal{Q}$, β, *semScores*, k, δ

```
1: aggLB = [], aggUB = [], resolution = 0, origSize = |T|
2: repeat
3:     resolution = getNextResolution(resolution, k, |T|)
4:     r_lowres = sample(Q.r, resolution)
5:     for all i, (l, s) ∈ T do
6:         s_lowres = sample(Q.r, resolution)
7:         scoreLB, scoreUB = signalSimCI(r_lowres, s_lowres)
8:         aggLB[i] = getTopFScores(scoreLB, semScores[i], β, k)
9:         aggUB[i] = getTopFScores(scoreUB, semScores[i], β, k)
10:    end for
11:    thresLB = getLowerBoundThres(aggLB, k)
12:    T' = []
13:    for all i, sim ∈ aggUB do
14:        if sim ≥ thresLB then
15:            T'.append(T[i])
16:        end if
17:    end for
18:    T = T'
19: until |T'| / origSize < δ
20: return T'
```

4.1.1 Confidence Interval Signal Pruning

To address the cost of calculating hundreds of millions of signal similarity scores for a given query, we use a form of top-k confidence interval pruning to eliminate candidates without calculating their exact score, an approach used when answering top-k queries over probabilistic data [32] and for recommending data visualizations over large datasets [36].

The core idea is to first quickly compute an approximate score using a *lower resolution* version of the data. A set of approximate scores, even with large error bounds, is often sufficient to disqualify many low-scoring candidates from further consideration. For the few potentially high-scoring candidates that remain, the system uses additional data to reduce the error bounds until it obtains the top-k topics.

Algorithm 3 details our pruning method. The input includes the topic database $\mathcal{T}$; user query $\mathcal{Q}$ and weighting preference β; the *semScores* for each topic in $\mathcal{T}$; the target size of $\mathcal{R}$ (k); and the desired pruning ratio (δ). On lines 3–6, the system creates low-resolution versions of the query signal r and each topic in $\mathcal{T}$ using a *resolution* value chosen by examining the size of set $\mathcal{T}$, k, and the previous *resolution* value (further discussed in Section 4.2). Then, the system calculates approximate signal similarity scores with 95% confidence intervals for each $(r_{\text{lowres}}, s_{\text{lowres}})$ pair. Using these intervals, the system finds lower and upper bounds on the harmonic mean scores (lines 7–9). The k-th highest lower bound in the harmonic mean scores becomes the pruning threshold (line 11); RaccoonDB prunes all topics with upper bounds below the threshold to produce $\mathcal{T}'$ (lines 12–17). The pruning repeats until the pruning ratio is less than δ.

4.1.2 Semantic Scoring Pruning

We can also use a pruning approach to avoid work during semantic scoring. We note that the size of a typical thesaurus is relatively small, and it is easy to precompute Jaccard similarities for all token pairs with non-zero similarity.

We exploit this fact in our semantic pruning (Algorithm 4). Like signal pruning, semantic pruning has $\mathcal{T}$, $\mathcal{Q}$, and k as input. Additionally, it requires a threshold (*thres*) that controls the aggressiveness of the pruning. On lines 1–2, the algorithm uses a precomputed data structure to retrieve tokens that have a Jaccard similarity with any token in the query higher

Algorithm 4 Semantic Pruning

Input: $\mathcal{T}$, $\mathcal{Q}$, *thres*, k

1: $tokens$ = getTokensAboveThres($\mathcal{Q}.q$, *thres*)
2: $candidates$ = getTopics($\mathcal{T}$, $tokens$)
3: $\mathcal{T}' = []$
4: **for all** $(t, s) \in candidates$ **do**
5: **if** jaccardScore(t, $\mathcal{Q}.q$) $\geq$ *thres* **then**
6: $\mathcal{T}'$.append((t, s))
7: **end if**
8: **end for**
9: return $\mathcal{T}'$

than *thres*; topics containing any of these tokens are then retrieved. Next, the full Jaccard score is calculated for the retrieved topics, and any topics with a Jaccard score below *thres* are pruned (lines 3–8). Controlling the value of *thres* over repeated calls is a crucial part of the algorithm and is described below in Section 4.2.

One guarantee we get from our thesaurus scoring is that our semantic pruning will always return *all* of the topics with a score greater than or equal to our threshold. As Algorithm 2 describes, the final similarity score is essentially the average across all token similarities. Therefore, if a topic has a score greater than or equal to *thres*, it must have a token with a Jaccard score greater than or equal to *thres*. Otherwise, the average of the Jaccard scores would be below *thres*.

4.2 Dynamic Query Optimizations

Algorithm 5 shows our full algorithm for optimized nowcasting query processing, which combines signal and semantic pruning, and dynamically adjusts pruning parameters based on $\mathcal{Q}$ and β. The core idea is that each pass through the main loop of the algorithm (lines 3–12) uses incremental amounts of semantic and signal information to eliminate candidates for $\mathcal{R}$, until there are only k candidates remaining.

For many queries, RaccoonDB could simply perform the pruning in Algorithms 3 and 4 with static parameters (e.g., *thres* = 0.5). Other queries, though, benefit from different parameter settings. For example, queries that skew topic semantic scores to high values should use a higher threshold to avoid excess work in semantic pruning.

When first processing a query, RaccoonDB decides an initial value for *thres* (lines 1–2): given the user's query string q, RaccoonDB creates a histogram of Jaccard scores to find a value of *thres* such that the number of candidate items is greater than or equal to N, which is chosen based on past query processing statistics. If N is too large, many items will be unnecessarily processed due to inadequate pruning; if it is too small, too many items will be pruned and extra rounds of query processing will be needed. As hinted at above, the histogram can vary greatly for each query, and one that has a distribution skewed towards higher or lower values will respectively benefit from lower and higher *thres* values.

After the first round of semantic pruning and scoring (lines 4–5), RaccoonDB applies signal pruning and scoring (lines 6–7). In this phase, since each iteration has relatively expensive sampling and data copying costs, RaccoonDB attempts to minimize the number of iterations needed to meet the stopping criteria on line 19 of Algorithm 3 by dynamically adjusting the *resolution* value. A query signal that correlates well with many topics benefits from a higher *resolution*.

Once scoring is complete (lines 4–7), RaccoonDB uses the k-th best item's score to find the minimum semantic score that satisfies Equation 1 with a perfect signal score of 1.0. This becomes our minimum Jaccard score (*minJacScore*),

Algorithm 5 Query Processing with Optimizations

Input: $\mathcal{T}$, $\mathcal{Q}$, β, k, δ

1: $minJacScore = +\infty$
2: *thres* = getNewThres($\mathcal{Q}$, β, $minJacScore$)
3: **while** *thres* $<$ $minJacScore$ **do**
4: $\mathcal{T}'$ = semanticPruning($\mathcal{T}$, $\mathcal{Q}$, *thres*, k)
5: $semScores$ = computeSemScores($\mathcal{T}'$, $\mathcal{Q}$)
6: $\mathcal{T}''$ = signalPruning($\mathcal{T}'$, $\mathcal{Q}$, $semScores$, β, k, δ)
7: $sigScores$ = computeSigScores($\mathcal{T}''$, $\mathcal{Q}$)
8: $\mathcal{R}$ = getTopFScores($semScores$, $sigScores$, β, k)
9: $kthBest = \mathcal{R}[k].score$
10: $minJacScore = (kthBest * \beta^2)/(1 + \beta^2 - kthBest)$
11: *thres* = getNewThres($\mathcal{Q}$, β, $minJacScore$)
12: **end while**
13: $factors$ = PCA($\mathcal{R}$)
14: (A, o) = factorCombiner($factors$, $\mathcal{Q}$)
15: $isLowQuality$ = queryQualityAlarm($\mathcal{Q}$, A, o)
16: return $(A, o, isLowQuality)$

which determines a new Jaccard threshold (lines 9–11). This Jaccard threshold is guaranteed to be monotonically increasing since at each round of pruning, only higher scoring items are included (as described in Section 4.1.2). Finally, after one or more rounds of pruning occur, the rest of the pipeline continues as in the unoptimized system (lines 13–16).

4.3 Low-Level Optimizations

There are several low-level database system optimizations available to RaccoonDB for all user queries. First, we can represent signal, semantic, and harmonic mean scoring (lines 4–8 of Algorithm 5) as vector operations, enabling us to use SIMD instructions. Second, by using a compact integer encoding for token labels, we can speed up lookups on lines 1–2 of Algorithm 4. Third, by partitioning $\mathcal{T}$, we can run lines 1–8 of Algorithm 5 in parallel on multiple CPUs.

4.4 Time Complexity

As discussed in Section 3.1, our naïve query processing runs in $O(n \log(n))$ worst-case time. However, our pruning methods reduce the amount of candidate topics that are sorted from n to p. With the $O(n)$ scoring pass over the data, our optimized system runs in $O(n + p \log(p))$ time. In most reasonable cases, $p \ll n$, so RaccoonDB processes queries in $O(n)$ time (i.e., the scoring time). In rare cases where pruning is ineffective (that is, p is not significantly smaller than n), the worst case case running time is $O(p \log(p))$.

5. DETECTING POOR USER QUERIES

As discussed in Section 2, RaccoonDB supports two query models. In the standard model, the user's query signal r is historical ground truth data about her target phenomenon, while in the distant supervision query model, r is whatever domain knowledge she has about her target's trend. In the latter query model, even if the user is a domain expert, it may be easy for her to unintentionally provide a low-quality version of r. Knowing that unemployment claims, for example, peak after the winter holidays allows the user to define a signal peak in January, but the amplitude and exact location on the timeline may be far from the truth.

The distant supervision query model can provide a user with valuable nowcasting results that have been heretofore impossible, but when the user's query signal r is unknowingly wrong, the nowcasting result may also be wrong without the

user realizing it. In this section, we propose a method of detecting when r is likely to be far from the ground truth when *no ground truth is available for comparison*, so the user can reevaluate her query before accepting a result.

Query Consistency — When a user's query contains no conventional ground truth data, our system cannot quantitatively evaluate how well its estimate of the target phenomenon matches reality. We can, however, evaluate properties of the user's query to judge the likelihood that the query will produce a high-quality estimate. More specifically, the query's two components (q, r) must be mutually consistent; that is, topics satisfying the semantic portion of the query should be related to the topics that satisfy the signal portion, and vice versa. If the two query components are unrelated, the nowcasting output of the system will be unrelated to at least one of the query inputs, which would qualify as a low-quality nowcasting result.

The quality of the user's query string q is fairly easy for a user to judge for herself (i.e., "Does 'I need a job' indicate unemployment?" should be an easy question for most users to answer). The query signal r, conversely, is difficult to evaluate without access to ground truth data. To assist the user, RaccoonDB uses a *query quality alarm* to examine her query to see if it is likely to produce a poor-quality result.

Building a Query Quality Alarm — The query quality alarm is a domain-independent classifier that predicts whether or not a query will produce a poor nowcasting result. The classifier uses the following groups of features to characterize important properties of the data:

- *Query Consistency* — These two features, f_{sem} and f_{sig}, measure the consistency of the two query components, as described above. To do this, the system finds the top-v topics as measured by signal similarity and computes the mean semantic similarity for them, and vice versa. For top signal topics S and top semantic topics T:

$$f_{\text{sem}} = \frac{1}{v}\sum_{s\in S} semScore(s) \qquad f_{\text{sig}} = \frac{1}{v}\sum_{t\in T} sigScore(t)$$

- *Variance Explained* — By applying PCA to a set of signals, one can measure the variety among the signals by looking at the resulting eigenvalues. RaccoonDB uses this observation to generate two features: the variance explained (i.e., eigenvalue) by the first PCA factor for the top-k topics as measured by signal similarity, and vice versa using semantic similarity.
- *Consistency Across β Settings* — To compute another series of features that indicate query consistency, the system finds the top topics as measured by one of our similarity metrics (e.g., signal similarity) and uses Equation 1 to find the average harmonic mean score (F_β) for the topics. We repeat this for both of our similarity metrics and β settings of 0.25, 0.5, 1, 2, and 4.

We use a random forest classifier built using the scikit-learn toolkit [31] and trained with past queries submitted to the system, allowing the system to improve as usage continues over time.[3] We label training queries as high quality if RaccoonDB's output scores greater than a user-defined threshold (we use 0.5 in our experiments) in both semantic relevance and signal correlation and as poor quality, if not.

[3]When being brought online initially when no query history exists, one can "prime" the alarm with conventional datasets.

Phenomenon	Description
boxoffice	US movie box office returns [1]
flu	US reported flu cases [2]
gas	US average gasoline price per gallon [6]
guns	US gun sales [3]
temp	New York City temperature [7]
unemployment	US unemployment insurance claims [5]

Table 2: Target phenomena used in experiments. All targets are measured weekly, except guns and temp, which we respectively convert from monthly and daily to weekly.

6. EXPERIMENTS

In this section, we present our evaluation of our four main claims about RaccoonDB's performance and quality:

1. The optimizations in Section 4 allow RaccoonDB to process queries in interactive time (Section 6.2).
2. The standard query model helps users with historical ground truth data generate *relevant* and *accurate* estimates of various real-world phenomena (Section 6.3).
3. The distant supervision query model helps users *without any ground truth data* generate *relevant* and *accurate* estimates of various real-world phenomena (Section 6.4).
4. The query quality alarm in Section 5 can accurately detect low-quality queries (Section 6.5).

6.1 Experimental Setting

Target Phenomena — We evaluated RaccoonDB using six publicly available datasets (Table 2). We chose these datasets because: (1) their availability of ground truth data; (2) their use in past nowcasting research [10, 11, 14, 17]; and (3) they describe a good mixture of both seasonal (e.g., flu) and non-seasonal phenomena (e.g., guns). Additionally, we evaluated RaccoonDB using a dozen other target phenomena: those targeted by Google Finance's Domestic Trends [4]. These latter targets are discussed in Section 6.4.

Social Media Data — Our experiments were run against a corpus of 40 billion tweets, collected from mid-2011 to the beginning of 2015. This represents roughly 10% of the Twitter stream. We filtered out non-English messages using a trained language classifier that shows 98% precision.

Topic Extraction — We extracted topics using a simple string containment approach as described in Section 2. We first applied "commodity" natural language preprocessing: removing obscure punctuation, replacing low-frequency tokens with generic type-based tokens (e.g., URLs become $<URL>$), and normalizing capitalization. We then populated $\mathcal{T}$ by extracting n-grams from each message, enumerating every consecutive sequence of four or fewer words (e.g., the message "Merry Christmas" would be associated with three topics: *merry*, *christmas*, and *merry christmas*). We then removed all topics with fewer than 150 associated messages. Instead of raw counts for the signal data, we use the counts normalized by the size of $\mathcal{M}$.

System Configuration — We ran our RaccoonDB and PostgreSQL experiments on a 32-core (2.8GHz) Opteron 6100 server with 512GB RAM, and we ran our Apache Spark experiments using Amazon c3.8xlarge EC2 instances.

6.2 Performance Evaluation

Summary: RaccoonDB answers nowcasting queries orders of magnitude faster than nowcasting with popular data management tools (PostgreSQL and Apache Spark). Our

Target	1 core			30 cores		
	Post-greSQL	NonOp	FullOp	Spark	NonOp	FullOp
boxoffice	> 6 h	909 s	6.2 s	1141 s	62.0 s	1.5 s
flu	> 6 h	935 s	1.4 s	1155 s	62.6 s	0.6 s
gas	> 6 h	1003 s	1.6 s	1157 s	65.9 s	0.5 s
guns	> 6 h	959 s	1.4 s	1173 s	64.7 s	0.8 s
temp	> 6 h	1003 s	3.5 s	1197 s	66.1 s	1.1 s
unemp	> 6 h	959 s	7.6 s	1215 s	68.2 s	2.5 s
Average	> 6 h	962 s	3.6 s	1173 s	64.9 s	1.2 s
FullOp Speedup	> 10^4x	424x	-	1356x	75x	-

Table 3: Query processing times for PostgreSQL, Apache Spark, and two versions of RaccoonDB: one without our pruning optimizations (NonOp) and another with them enabled (FullOp).

optimizations allow for interactive speed processing, with a **424x** speedup over a non-optimized version of RaccoonDB.

Overview — In this experiment, we measured the runtime of several nowcasting query processing systems. An ideal system would return results in interactive time so that users can quickly iterate through revisions toward their final result.

Evaluation Metrics — We evaluated the performance of RaccoonDB by measuring the query processing time for each of our target phenomena. Additionally, since our pruning optimizations eliminate candidates in part based on a 95% confidence bound—potentially removing some high-quality candidates—we also measured our optimized system's recall of $\mathcal{R}$ compared to a non-optimized version of RaccoonDB.

Baselines — We first measured how well two popular data systems, PostgreSQL and Apache Spark, can be used for declarative nowcasting. For PostgreSQL, we stored our topics in a table and iterate over them using the built-in `CORR()` function for signal scoring and our own UDF for semantic scoring. For Apache Spark, we tested two clusters of Amazon EC2 c3.8xlarge instances (each using 30 cores), the first with 1 node, the other with 10 nodes. The Spark-based system was written in Python, and it preloaded topics into memory and cached intermediate data between job stages.

Since PostgreSQL and Spark are more generalized systems, we also compared RaccoonDB against a "non-optimized" version of itself (NonOp), which lacks the optimizations discussed in Sections 4.1 and 4.2, but does include the low-level optimizations in Section 4.3 (vectorized operations, parallelization, etc.), allowing us to measure the impact of our core optimizations. NonOp, along with the optimized system, were written in C++, with topics and optimization-based data structures preloaded into memory.

Overall Performance Results — As Table 3 shows, our optimized system (FullOp) averaged just 3.6 seconds per query on a single processor core, achieving interactive query processing times with very few resources. In contrast, PostgreSQL was far from efficient, not finishing even after 6 hours (at which point we ended the evaluation). RaccoonDB itself is far less efficient without its optimizations: using a single core, FullOp yielded a **424x** average speedup over NonOp, which averaged 962 seconds (about 16 minutes) per query—far from interactive time.

If we allow for parallelization, systems like Apache Spark may appear to be reasonable choices; however, when using 30 cores (on one node), Spark averaged 1173 seconds (about 20 minutes) per query. In contrast, FullOp averaged just 1.2 seconds per query using 30 cores, with a **1356x** average speedup over Spark. Increasing the Spark cluster size to 300 cores (over 10 nodes) reduced the average runtime to 106.8 seconds, but our 30-core FullOp system was still on average **123x** faster *using only 10% of the computational resources.*

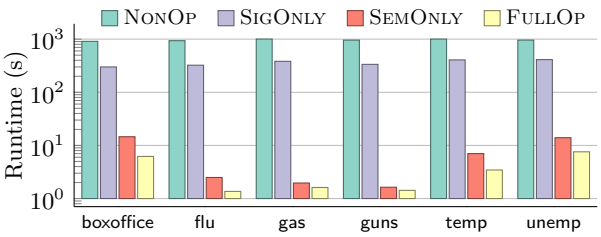


Figure 2: Effects of different combinations of RaccoonDB's pruning optimizations. Results shown are for $\beta = 1$ on 1 core; runtime in logarithmic scale.

	NonOp		FullOp	
Semantic Scoring	214.6 s	(22%)	0.97 s	(27%)
Signal Scoring	737.8 s	(77%)	1.16 s	(32%)
Rank Aggregation	9.2 s	(1%)	0.11 s	(3%)
Optimization Overhead	*n/a*	(0%)	1.37 s	(38%)
Total	961.6 s		3.61 s	

Table 4: Breakdown of 1-core runtime for NonOp and FullOp, averaged over all targets. Overhead includes pruning in FullOp.

To measure the direct benefit of our optimizations on RaccoonDB with parallelization, we enabled parallelization on our non-optimized system and compared it against our optimized system. Using 30 cores, the non-optimized system averaged 65 seconds per query, still too long to allow for interactive, human-in-the-loop style querying. In contrast, our optimized system's 1.2 second average runtime resulted in a **75.2x** speedup over our 30-core non-optimized system.

Impact of Pruning on Recall — Unlike RaccoonDB's semantic pruning, the confidence interval-based signal pruning is lossy; the 95% confidence interval used for pruning may cause the pruning away of high-quality topics. We can measure the impact of lossy pruning by measuring the recall of the optimized system's $\mathcal{R}$ against the non-optimized $\mathcal{R}$. In the experiments for Table 3, RaccoonDB achieved 100% recall with signal pruning (i.e., *never* losing a single topic).

Impact of Pruning on Runtime — Figure 2 shows the impact of our different pruning optimization methods on runtime using a single core. With no optimizations (NonOp), query processing took 962 seconds on average for our target phenomena. Using signal pruning alone (SigOnly) reduced runtime by 62.5% to 360 seconds (6 minutes) on average. Using semantic pruning alone (SemOnly) reduced NonOp's runtime by 99.3% to 7 seconds. Using both pruning techniques together further reduced the runtime to 3.6 seconds—a 48% reduction from SemOnly and a 99.6% reduction overall.

Semantic pruning was so effective because it is able to cheaply identify items that can be eliminated, removing over 99% of topics on average for our target queries. Signal pruning eliminated a similar percentage of topics, but because it requires semantic scores be computed, using signal pruning alone resulted in a much smaller computational savings.

Table 4 shows how the non-optimized (NonOp) and optimized (FullOp) versions of RaccoonDB spent their time (in this case, using 1 core). NonOp spent the majority of its time scoring, plus a small amount on combining the scores.

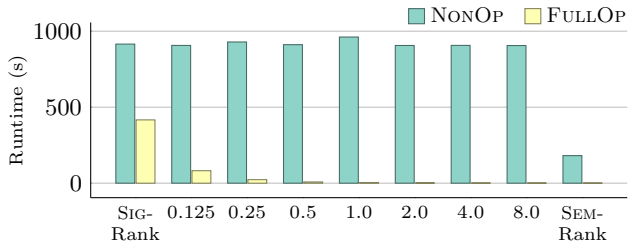


Figure 3: Runtimes for the non-optimized (NonOp) and optimized (FullOp) systems with different values of β on 1 core.

For FullOp, a significant portion of its total runtime was due to pruning overhead. Of course, the absolute runtimes for the optimized system are drastically smaller.

Impact of β on Runtime — As discussed in Section 4, there is an interaction between β and our optimizations. In this experiment, we measured how this interaction affects runtime. Figure 3 shows the results of varying β for RaccoonDB (with 1 core), averaged over all nowcasting queries. When β weights signal scores more heavily, our semantic pruning had a smaller effect. In the extreme case of SignalRank, in which the *semScores* do not impact the final output at all, nothing can be pruned semantically. Fortunately, this is an unusual use case of RaccoonDB, where the goal is generally to use both query components to select topics. However, these runtimes can be improved to interactive time by increasing the number of cores.

6.3 Quality of Standard Query Model

Summary: Compared to our baseline methods of users manually searching for topics or ranking them by similarity scores, results generated with RaccoonDB's standard query model require far less work from users and are of higher quality.

Overview — In this experiment, we measured the result quality from RaccoonDB and our baselines performing a traditional nowcasting task in which high-quality historical data is available to the user. An ideal result should closely estimate the target phenomenon and should be generated from relevant topics. An effective nowcasting query system should be able to beat our baselines (see below) for at least some setting of β, and ideally should do it for many settings.

Evaluation Metrics — We evaluated signal quality using a standard nowcasting evaluation model [10]: using the available social media signals at time t, we generated a statistical model to estimate a value at time $t+1$; then, the updated social media signals at time $t+1$ are used to build a new model and estimate for $t+2$. This process repeats for the data's entire time period, starting one year into the dataset. Pearson correlation is then used to evaluate these forecasts against the ground truth data for the same time period.

In order to measure semantic relevance, we asked a set of human evaluators (via Amazon Mechanical Turk) to rate topic labels as *relevant* or not when applied to a given nowcasting query. Evaluators rated 10 topic labels for a single query—chosen randomly from the top-100 $\mathcal{R}$ results returned by RaccoonDB—with the majority vote from five evaluators deciding if the topic label is relevant. The semantic relevance for a nowcasting result is the normalized number of topic labels that were marked as relevant.

Baselines — We compared RaccoonDB to several baseline approaches that we created to simulate the workflow

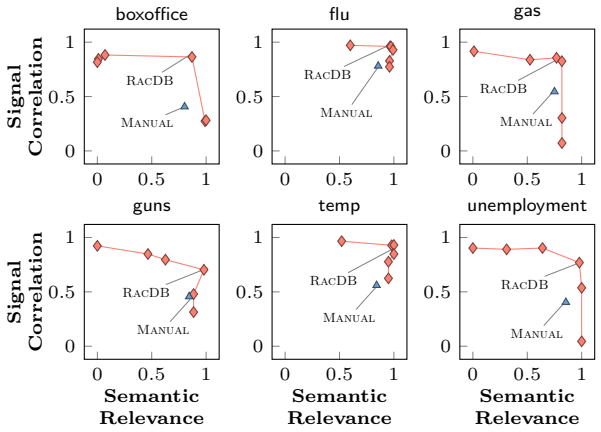


Figure 4: RaccoonDB (RacDB) evaluated on nowcasting for $\beta = \{0, 0.125, 0.25, 1, 128, \infty\}$ with our user study baseline (Manual); SignalRank ($\beta = 0$) and SemanticRank ($\beta = \infty$) are the end points on the RacDB line.

Method	Standard Query Model			Dist. Supervision Query Model		
	Sig. Corr.	Sem. Rel.	SS-F_1	Sig. Corr.	Sem. Rel.	SS-F_1
RaccoonDB	0.82	0.94	**0.88**	0.67	0.97	**0.79**
Manual	0.53	0.80	0.64	0.53	0.80	0.64
SemanticRank	0.35	0.94	0.51	0.35	0.94	0.51
SignalRank	0.92	0.19	0.31	0.79	0.06	0.11

Table 5: Nowcasting quality results for RaccoonDB's two query models and our baselines. **SS-F_1** is the harmonic mean of the signal correlation and semantic relevance scores of each method. All values range from 0–1, with 1 being best.

found in existing nowcasting literature. For the first baseline (Manual), we performed a user study designed to mimic the interactive string-picking procedure followed by most past nowcasting projects. We asked eleven technically sophisticated users to build a nowcasting model for our six target phenomena. We gave users a search tool that would take a topic label as input and then display the corresponding topic signal (assuming it was found in $\mathcal{T}$). After each search, we also displayed a composite signal, which our system generated by following the aggregation procedure described in Section 3. We allowed the users to add or remove up to fifteen topics without any time limit. We chose each user's best result for each nowcasting target and we averaged the signal relevance and semantic correlation scores of all users.

For the next two baselines, we mimicked the nowcasting workflow of ranking candidate topics by a similarity metric. The first of these (SemanticRank) is an "all semantics" method that uses only semantic-based ranking of topics from a nowcasting query string (e.g., [12]). The second (SignalRank) is an "all signal" technique that ranks topics based on their signal correlation with a nowcasting query signal (e.g., [17]). RaccoonDB can emulate SemanticRank and SignalRank by setting β to extreme values ($\beta = 0, \infty$).

Overall Results — Figure 4 and Table 5 summarize these results. On the x-axis of each plot is the semantic relevance (measured by human judges), and on the y-axis is the signal correlation (measured by Pearson correlation on held-out ground truth signal). RaccoonDB results are shown as a line to indicate the possible answers RaccoonDB provides based on different β parameter values. On these figures, we display RaccoonDB results for $\beta = \{0, 0.125, 0.25, 1, 128, \infty\}$, with

the extreme settings being our baselines SEMANTICRANK ($\beta = 0$) and SIGNALRANK ($\beta = \infty$). Our user study baseline (MANUAL) is shown as an individual point.

Not surprisingly, MANUAL and SEMANTICRANK had great semantic relevance (on average, 0.83 and 0.94), but their signal correlation was mediocre for many targets (flu was an exception), averaging 0.53 and 0.35. In contrast, SIGNALRANK had high signal correlation (0.92), but the semantic relevance was very low (0.19), indicating that the high signal correlation was likely due to spurious statistical correlations with the user query signal r. Conversely, RACCOONDB performed well on both signal correlation and semantic relevance (0.82 and 0.94), with results that dominate MANUAL and often SEMANTICRANK and SIGNALRANK as well. More balanced β settings (i.e., closer to 1) generally performed best.

The strongly seasonal targets like flu and temp were easier to estimate with RACCOONDB and the baselines, while other less-seasonal targets like guns were harder to estimate accurately. On average, if we set $\beta = 1$, RACCOONDB used relevant topics and correlated quite well with the target signal, correlating **57%** better than MANUAL and **135%** better than SEMANTICRANK. While SIGNALRANK correlated well with the ground truth signal, its topics had virtually no semantic relevance, so a user would not accept its results.

6.4 Quality of Distant Supervision Model

Summary: RACCOONDB can reasonably estimate real-world phenomena using no ground truth data, requiring far less work and producing higher-quality results than our baseline methods. Further, with a few query revisions, RACCOONDB results can improve significantly.

Overview — As mentioned in Section 2, it is often possible for researchers to encode domain knowledge (*unemployment follows Christmas* or *movies are popular in summer*) in a hand-made distantly supervised signal. The obvious risk here is that providing RACCOONDB with low quality signal data will yield low-grade or misleading nowcasting results.

To test this scenario, we evaluated RACCOONDB with the same target phenomena described in Section 6.1, but we replaced the query signal r with a degraded replacement meant to emulate crude user domain knowledge. We transformed each r into a low-quality r' by replacing each human-marked signal peak with a synthetic parameterized peak characterized by just one of four possible amplitude values, one of four widths, and a hand-chosen maximum.

Evaluation Metrics and Baselines — We used the same evaluation metrics and baselines as in Section 6.3.

Overall Results — As Table 5 shows, RACCOONDB's signal correlation dropped from 0.82 to 0.67 when going from the standard query model to the distant supervision model, but RACCOONDB still achieved a higher signal correlation than MANUAL (0.52) and SEMANTICRANK (0.35)—all methods with more than acceptable semantic relevance scores. As with the standard query model, SIGNALRANK achieved a high signal correlation (0.79), but the terrible semantic relevance (0.06) leads us to not trust this result.

Iterative Querying Quality — In this experiment, we evaluated how RACCOONDB result quality improves when users iteratively revise their queries. We assumed that users revise queries when the result has a semantic relevance below 0.5. For example, in an effort to improve the second query for

Target	MANUAL	SEM-RANK	SIG-RANK	RACDB	ITER-RACDB
Air Travel	**0.62**	0.00	0.06	0.37	-
Auto Finance	0.28	0.00	0.29	0.48	**0.76**
Bankruptcy	0.45	0.00	0.47	0.25	**0.50**
Construction	0.41	0.02	0.61	0.79	-
Credit Cards	0.62	0.00	0.49	**0.73**	
Dur. Goods	0.03	0.00	0.12	**0.81**	-
Education	0.83	0.72	0.65	**0.86**	-
Furniture	0.36	0.00	0.42	**0.45**	-
Insurance	0.12	0.00	0.25	0.41	**0.69**
Mobile	**0.92**	0.35	0.86	0.88	-
Real Estate	0.04	0.08	0.19	0.51	**0.70**
Shopping	0.27	0.04	0.07	**0.92**	-
Average	0.41	0.10	0.37	0.62	**0.71**†
Std. Dev.	0.28	0.21	0.24	0.22	**0.17**†

Table 6: RACCOONDB result quality improvements after revising queries with poor semantics (ITERRACDB). Values are the harmonic mean of semantic relevance and signal correlation. †These statistics include RACDB results if no revisions were made.

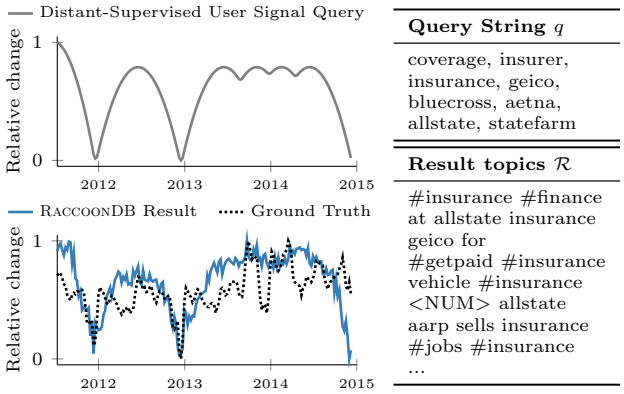


Figure 5: A distant supervision query and nowcasting result for the target phenomenon of *Insurance* (using ITERRACDB).

Insurance, we added several insurance company names (e.g., *Aetna* and *Geico*). Since users can easily identify irrelevant topics, this is more analogous to RACCOONDB's real-world usage. We tested this using 12 new target phenomena that lack ground truth data (chosen from Google Domestic Trends [4]). For each target, we generated queries with synthetic signals (as described above), and we measured signal correlation with the original Google Trends estimate. We used the same baselines as above, except for MANUAL, where we use the keywords listed by Google. Figure 5 shows the query and result for *Insurance* after one round of revising.

As Table 6 shows, RACCOONDB with only one round of querying (RACDB) generated higher-quality results than our baselines for 9 out of 12 targets, with an average harmonic mean between semantic relevance and signal correlation of 0.62. When allowing for query revisions (ITERRACDB), result quality improved to 0.71, beating our baselines on 10 out of 12 targets and achieving a 73% increase on average over using the Google-chosen keywords (MANUAL). For consistency, these results used $\beta = 1.0$; however, some targets had higher-quality results with a different weighting preference (e.g., *Air Travel* with $\beta = 2.0$ had a harmonic mean score of 0.63). RACCOONDB allows users to explore these different results.

Additional Results — To explore RACCOONDB's capabilities in light of low-quality data, we introduced varied levels of error into our original distant supervision queries (boxoffice, etc.). RACCOONDB still did surprisingly well. Due to limited space, these results are presented in a separate article [8].

Alarm Model	Semantic Relevance	Signal Correlation	Harmonic Mean Score
None	0.57	0.35	0.43
Alarm	0.61 (5.4%)	0.45 (30.4%)	0.52 (19.6%)
Alarm + User	0.84 (46.0%)	0.43 (24.2%)	0.57 (31.6%)

Table 7: Average result quality improvement from excluding queries that triggered the alarm (*Alarm*) and also excluding results with low semantic relevance (*Alarm + User*).

6.5 Effectiveness of Quality Alarm

Summary: Through a user study, we show that our query quality alarm is able to identify low-quality queries, improving average result quality by 31.6%.

Overview — In this experiment, we evaluated how well our query quality alarm can identify queries that produce low-quality results. The failure mode our alarm guards against is in our distant supervision query model when the user provides a very poor query signal r without intending to, and as a result, gets a very surprising query result. To test our alarm, we collected a labeled training set from a user study, then we used leave-one-out cross validation, training on all but one target phenomenon for each round of evaluation. We presented each of our 18 study participants with each of our target phenomena and asked them to provide the query tuple (q, r); they used an online drawing tool to provide a distantly supervised version of r. To allow us to train and test our classifier on a substantial number of user queries, we synthesized new queries by taking the Cartesian product of the query strings and signals provided by our 18 users. This method allowed us to create an additional 306 synthetic queries for each of the 6 nowcasting targets.

Evaluation Metrics — We measured the average improvement of semantic relevance and signal correlation before and after filtering out alarmed results. We also measured precision and recall of our alarm's ability to identify low-quality queries. To simplify our analysis, we used $\beta = 1$ for all queries; however, it is worth noting that for some queries, a different β value achieved higher-quality results.

Overall Results — Table 7 shows the result of applying the query quality alarm in two different settings, with each result averaged over our six target phenomena. *None* shows the average quality across all queries. *Alarm* shows the results for those queries that did not trigger the alarm, increasing average signal correlation 30.4% (from 0.35 to 0.45).

In the second setting, *Alarm + User*, we assumed that users perform an additional step of pre-filtering results with a semantic relevance below 0.5. Since users can easily identify irrelevant topics, this is more analogous to RaccoonDB's real-world usage. The signal correlation increased to 0.43 (a 24.2% increase over not using the alarm). While this is not quite as large an increase as using the alarm alone, this method did show a significant increase in semantic relevance. This simple extended alarm had a precision of 0.74 and a recall of 0.89, with a false positive rate of 16% over all of the targets. For all targets, both the average semantic relevance and signal correlation scores showed improvement.

7. RELATED WORK

There are several areas of research that are related to our work, which we discuss below.

Nowcasting — Several research projects [12, 14, 17, 34], including our own work [10], have used ad hoc social media nowcasting to estimate real-world phenomena, relying on either slow and tedious manual searching or automated statistical approaches that are prone to spurious and low-quality results. We extend this work by proposing a general-purpose declarative nowcasting query system that quickly generates high-quality results with little user effort. Our work is also similar to Google Trends [14]—where users can search for social media topics by keyword—and Google Correlate [29]—where users can rank topics by signal correlation. However, in contrast to our system, neither take advantage of both criteria at the same time. Our work is also distinct from Socioscope [38], which assumes relevant topics have already been identified. While nowcasting has faced some criticism due to model failures [24] and the cherry-picking of results [25], we view nowcasting as a tool that can be used either effectively or maliciously—like statistics in general—and with proper domain knowledge and training, these issues can be avoided.

At the end of Section 1, we discuss how our current work differs from our past nowcasting work [9–11].

Feature Selection — Feature selection is a well-studied problem in the statistics and machine learning domains. Guyon and Elisseeff have a survey on the subject [18], where they discuss different classes of selection methods such as using domain knowledge, ranking methods, and dimensionality reduction techniques. In our work, we borrow ideas and methods from all three of these categories. More recently, Zhang et al. [40] introduce a system for fast interactive feature selection. We share their motivation of improving human-in-the-loop feature selection, but our semantic-based approach to scoring candidates takes this a step further by automating much of the human-driven relevance evaluation. Additionally, much of their work focuses on combining selection with model construction, but the scale of our topic database requires RaccoonDB to use a ranking-based selection process.

Query Optimization — Query optimization is a well-studied problem in database literature and has been applied to other domains, though standard RDBMS optimization techniques do not apply in our setting. Optimizations based on confidence interval pruning have been implemented in a number of works, including top-k query processing over probabilistic data [32] and in recommending visualizations for large datasets in the SeeDB project [36]. While similar to these other projects, our confidence interval signal pruning works in tandem with our novel semantic scoring pruning method, requiring dynamic parameter configuration on a query-by-query basis. Herodotou et al. [19] developed a system to identify good system parameters for MapReduce jobs, but they computed these separately from the actual job execution. RaccoonDB instead dynamically adjusts its parameters as the query is being executed.

Multi-criteria Ranking — The goal of multi-criteria ranking, or rank aggregation, is to combine multiple ranked orderings to produce a top-k list without evaluating all items. Rank-Join [21] and Fagin's algorithm [16] are two popular approaches. Our work also needs to achieve a top-k list with multiple scoring metrics; however, in our case, the scoring of candidates is most expensive, so we have to rely on a different set of optimizations (discussed in Section 4). Additionally, our work hints at a relationship to skyline queries [23]; however, our massive number of topics requires that we approximate the skyline by selecting one point from it (the top topic in a result) and aggregate it with $k - 1$ of its neighbors.

Query Formulation — This area of work focuses on using external information—such as query logs [22] or the data being queried [39]—to expand query results to include relevant items that user queries do not exactly describe. Our work shares a similar goal, but we can further improve results by evaluating them in their final usage in a nowcasting model.

Complex Event Processing (CEP) — CEP focuses on extracting trends from a stream of events, and could be used to identify topic-based signals; however, doing so would be extremely slow since CEP systems identify patterns across events directly (e.g., taking 30 seconds to process a simple "W" pattern match on 500K tuples [30]). Further, since they lack semantic-based topic selection, they would require the tedious manual topic selection that RaccoonDB avoids.

Sequential Pattern Mining — Note that our work is quite distinct from sequential pattern mining [27], in which the goal is to identify latent interesting events from time-stamped data. Rather, our task is to choose the most useful few sequences from a vast set of candidates.

8. CONCLUSION AND FUTURE WORK

In this work, we proposed a novel query model and framework for declaratively specifying a nowcasting model and yielding high-quality results with little user effort. Our query optimizations and quality alarm aid the human-in-the-loop nature of nowcasting by allowing for fast query iteration on many phenomena—even those lacking historical ground truth data. In the future, we would like to deploy RaccoonDB. We also see an opportunity for applying its principles to building other socially driven dataset construction tools.

Acknowledgements

This project is supported by National Science Foundation grants 0903629, 1054913, and 1131500, Yahoo!, and Google.

9. REFERENCES

[1] Box Office Mojo Weekly Box Office Index. `http://www.boxofficemojo.com/weekly/`, 2015.
[2] CDC Influenza Surveillance Data via Google. `http://www.google.org/flutrends/data.txt`, 2015.
[3] FBI - NICS. `http://fbi.gov/services/cjis/nics`, 2015.
[4] Google Domestic Trends - Google Finance. `http://www.google.com/finance/domestic_trends`, 2015.
[5] US Dept. of Labor - Unemployment Insurance Weekly Claims Data. `http://research.stlouisfed.org`, 2015.
[6] U.S. Gasoline and Diesel Retail Prices. `http://eia.gov/petroleum/gasdiesel/`, 2015.
[7] Weather Underground - Weather History New York City. `http://wunderground.com/history/airport/KNYC/`, 2015.
[8] M. R. Anderson, D. Antenucci, and M. Cafarella. Runtime support for human-in-the-loop feature engineering systems. *IEEE Data Engineering Bulletin*. To appear.
[9] D. Antenucci, M. R. Anderson, P. Zhao, and M. Cafarella. A query system for social media signals. In *ICDE*, 2016.
[10] D. Antenucci, M. Cafarella, M. C. Levenstein, C. Ré, and M. D. Shapiro. Using social media to measure labor market flows. Working Paper 20010, NBER, 2014.
[11] D. Antenucci, M. J. Cafarella, M. Levenstein, C. Ré, and M. Shapiro. Ringtail: Feature selection for easier nowcasting. In *WebDB*, 2013.
[12] N. Askitas, K. F. Zimmermann, et al. Detecting mortgage delinquencies. Technical report, IZA, 2011.
[13] J. Bollen, H. Mao, and X. Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2011.
[14] H. Choi and H. Varian. Predicting the present with Google Trends. Technical report, Google, Inc., 2011.
[15] K. W. Church and P. Hanks. Word Association Norms, Mutual Information, and Lexicography. *Computational Linguistics*, 16(1):22–29, 1990.
[16] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.
[17] J. Ginsberg, M. H. Mohebbi, R. Patel, L. Brammer, M. S. Smolinksi, and L. Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, February 2009.
[18] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR*, 3:1157–1182, 2003.
[19] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *SoCC*, 2011.
[20] L. Hong and B. D. Davison. Empirical study of topic modeling in Twitter. In *Workshop on Social Media Analytics*, 2010.
[21] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.
[22] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *PLVDB*, 4(1):22–33, 2010.
[23] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, 2002.
[24] D. Lazer, R. Kennedy, G. King, and A. Vespignani. The parable of Google Flu: Traps in big data analysis. *Science*, 343(6176):1203–1205, 2014.
[25] P. T. Metaxas, E. Mustafaraj, and D. Gayo-Avello. How (not) to predict elections. In *IEEE SocialCom*, 2011.
[26] D. Metzler, S. Dumais, and C. Meek. *Similarity measures for short segments of text*. Springer, 2007.
[27] M. Middelfart and T. B. Pedersen. Implementing sentinels in the TARGIT BI suite. In *ICDE*, 2011.
[28] M. Mintz, S. Bills, R. Snow, and D. Jurafsky. Distant supervision for relation extraction without labeled data. In *ACL-IJCNLP*, 2009.
[29] M. Mohebbi, D. Vanderkam, J. Kodysh, R. Schonberger, H. Choi, and S. Kumar. Google Correlate whitepaper. Technical report, Google, Inc., 2011.
[30] B. Mozafari, K. Zeng, and C. Zaniolo. From regular expressions to nested words: Unifying languages and query execution for relational and XML sequences. *VLDB*, 2010.
[31] F. Pedregosa et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
[32] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.
[33] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 2nd edition, 1979.
[34] S. L. Scott and H. Varian. Bayesian variable selection for nowcasting economic time series. In *Economics of Digitization*. University of Chicago Press, 2014.
[35] G. Tsatsaronis, I. Varlamis, and M. Vazirgiannis. Text relatedness based on a word thesaurus. *Journal of Artificial Intelligence Research*, 37(1):1–40, 2010.
[36] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. SeeDB: Efficient data-driven visualization recommendations to support visual analytics. In *VLDB*, 2015.
[37] L. Wu and E. Brynjolfsson. The future of prediction: How google searches foreshadow housing prices and sales. In *Economics of Digitization*. U. of Chicago Press, 2014.
[38] J.-M. Xu, A. Bhargava, R. Nowak, and X. Zhu. Socioscope: Spatio-temporal signal recovery from social media. In *ECML-PKDD*, 2012.
[39] J. Yao, B. Cui, L. Hua, and Y. Huang. Keyword query reformulation on structured data. In *ICDE*, 2012.
[40] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. In *SIGMOD*, 2014.