AggChecker: A Fact-Checking System for Text Summaries of Relational Data Sets

Saehan Jo[*], Immanuel Trummer[*], Weicheng Yu[*],
Xuezhi Wang[†], Cong Yu[†], Daniel Liu[*], Niyati Mehta[*]
Cornell University[*], Google Research[†]
{sj683, itrummer, wy248, dl596, nbm44}@cornell.edu, {xuezhiw, congyu}@google.com

ABSTRACT

We demonstrate AggChecker, a novel tool for verifying textual summaries of relational data sets. The system automatically verifies natural language claims about numerical aggregates against the underlying raw data. The system incorporates a combination of natural language processing, information retrieval, machine learning, and efficient query processing strategies. Each claim is translated into a semantically equivalent SQL query and evaluated against the database. Our primary goal is analogous to that of a spell-checker: to identify erroneous claims and provide guidance in correcting them. In this demonstration, we show that our system enables users to verify text summaries much more efficiently than a standard SQL interface.

PVLDB Reference Format:
Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, Niyati Mehta. AggChecker: A Fact-Checking System for Text Summaries of Relational Data Sets. *PVLDB*, 12(12): 1938-1941, 2019.
DOI: https://doi.org/10.14778/3352063.3352104

1. INTRODUCTION

Text summaries of relational data sets are very common. Typical examples include data journalism newspaper articles, company executive summaries of internal audits, and reports providing analyses of survey results. However, the correctness of numerical facts in these text documents is often taken for granted (i.e., left unchallenged), the claimed values seldom double-checked against raw data. In our prior work [6], we proposed AggChecker[1] to address this problem.

AggChecker is a system for fact-checking natural language claims about relational data. It takes as input a relational database and a text document containing claims about numerical aggregates derived from the database. The output of AggChecker is similar to that of a spell-checker. AggChecker provides a marked-up version of the input text where each claim is tentatively marked up as either correct or erroneous.

Example 1. Consider the following text *"There were only **four** previous lifetime bans in my database - **three** were for repeated substance abuse"* taken from a newspaper article [3]. It contains two claims (four and three) about the number of lifetime suspensions in the NFL. These claims are translated into the SQL queries SELECT COUNT(*) FROM NFLSUSPENSIONS WHERE GAMES = 'INDEF' and SELECT COUNT(*) FROM NFLSUSPENSIONS WHERE GAMES = 'INDEF' AND CATEGORY = 'SUBSTANCE ABUSE, REPEATED OFFENSE' on the associated data set. Our goal is to automatically translate text to queries, to evaluate those queries, and to compare the evaluation result against the claimed one.

Currently, we focus on claims that can be translated into queries of the form: SELECT FCT(AGG) FROM T1 E-JOIN T2 ... WHERE C1=V1 AND C2=V2 ..., an aggregate over an equi-join of tables with conjunctive equality predicates. Claims of this type are both popular and error-prone [6].

To verify a claim, AggChecker evaluates the corresponding query on the database and compares the query result with the claimed value. Thus, our main challenge is to correctly translate a claim (i.e., a natural language description of a numerical aggregate) into a semantically equivalent SQL query. Despite this similarity to natural language querying [7, 10], there are two particularities of fact-checking that introduce new opportunities. First, in fact-checking, we accept as input an entire document with multiple potentially related claims. We can exploit semantic relationships between different claims to facilitate claim-to-query translations. Second, since each claim contains a claimed result of its underlying SQL query, we can resolve ambiguity in natural language understanding by evaluating many query candidates against the claimed result. Prior work on automated fact-checking [5] focuses on checking claims against professionally-verified natural language facts. However, such verification results are only available for a small number of claims that received widespread attention.

AggChecker is based on a mixture of natural language analysis, information retrieval, machine learning, and efficient query evaluation strategies. In Section 2, we introduce the details of these components. In Section 3, we present a thorough description of our demonstration.

2. SYSTEM OVERVIEW

Figure 1 gives an overview of the AggChecker system. AggChecker starts with a text document and its associated relational data set as inputs. In the end, the system displays

[1] An online demo and a video tutorial of the system are available at `http://facts-demo-dbgroup.cs.cornell.edu/`.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352104

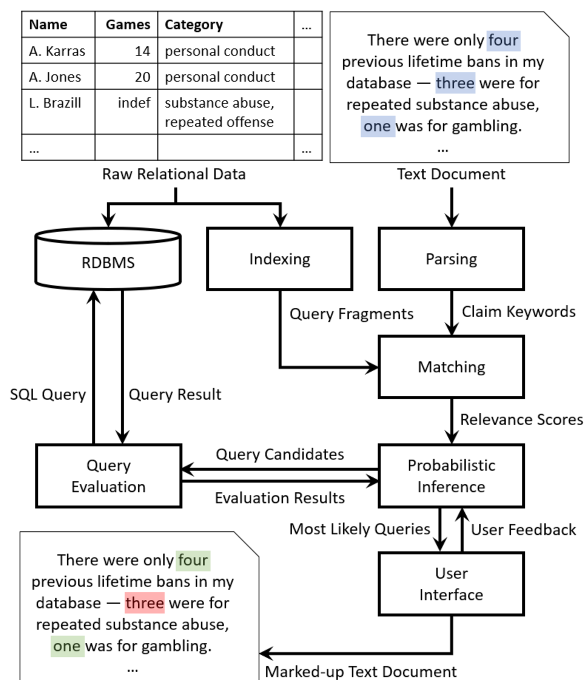


Figure 1: Overview of the AggChecker system.

its automated fact-checking result (i.e., marked-up version of input text) to users and integrates user feedback if necessary. The system can be decomposed into four main components: Indexing/Parsing/Matching, Probabilistic Inference, Query Evaluation, and User Interface.

Indexing/Parsing/Matching. The system produces keyword-based relevance scores for claims and query fragments. In the Indexing stage, the system indexes relevant keywords (e.g., column names, literal names, and their synonyms found in word dictionaries) for every query fragment that can be formed from the data set. In the Parsing stage, we associate each claim with a set of keywords using natural language processing (NLP) tools and word dictionaries. Finally, in the Matching stage, we use an information retrieval (IR) engine to query the keyword set of each claim against the keyword sets of query fragments. Consequently, we obtain a relevance score per query fragment-claim pair.

Probabilistic Inference. The heart of AggChecker is a probabilistic model which infers the probability distribution over SQL queries behind each claim. The primary challenge in fact-checking claims against relational data is to translate each natural language claim into a semantically equivalent SQL query. When translating claims to queries, ambiguities in natural language often create uncertainty. This motivates the use of a probabilistic model which can thoroughly reason over alternative query translations per claim. The model takes three inputs: 1) keyword-based relevance scores for each pair of a claim and a query fragment, 2) evaluation results of query candidates, and 3) user feedback. As a result, it outputs the likely SQL queries (and their probabilities) per claim. Then, the system tentatively marks up each claim as correct or erroneous based on the evaluation result of the most likely query.

Query Evaluation. AggChecker uses efficient query processing techniques like caching, query merging, and incremental execution to evaluate myriads of query candidates against an RDBMS. We compare the query results with the claim (i.e., claimed result in text) in order to obtain more information about the likelihoods of query candidates.

User Interface. Via an interactive user interface, we collect user feedback and adjust the probabilities of query candidates accordingly.

2.1 Indexing, Parsing, and Matching

Given a text document and its associated data set, our system calculates the keyword-based relevance scores between SQL query fragments and a claim. By query fragments, we denote an aggregation function, an aggregated column, or equality predicates which together can construct an aggregate query. The number of queries that can be formed from these query fragments is typically in the order of billions or trillions for typical cases.

First, the system considers all possible query fragments that can be generated based on the data set. We associate each query fragment with relevant keywords based on the names and descriptions of each contained database element. Likewise, we associate a claim with relevant keywords based on the document hierarchy, sentence structures, and synonyms, derived from the HTML markups, Stanford parser [8] and WordNet [2, 9], respectively. Keywords are weighted based on the document hierarchy and sentence structures. Note that it is often necessary to consider words outside the claim sentence to grasp the full context of a claim (increasing the translation accuracy by 7∼12%). Having associated a set of keywords with each claim and each query fragment, we use an IR engine, Apache Lucene [13], to calculate the relevance scores for query fragment-claim pairs. These scores form an input to the probabilistic model which infers the probabilities of query candidates being the underlying query behind a claim.

2.2 Probabilistic Inference

Given the relevance scores of SQL query fragments and natural language claims, AggChecker infers for each claim the most likely SQL queries. To achieve this goal, our system utilizes a probabilistic model and an algorithm based on expectation maximization (EM) to compute the probabilities of query candidates.

The system represents the main focus of a text document by prior probability distributions over SQL query fragments (p). In addition, there are two other factors that contribute to final probabilities of query candidates: 1) relevance scores from keyword matching (S_c) and 2) evaluation results of query candidates (E_c). Note that these factors are specific to each claim. Intuitively, a higher keyword-based relevance score of a query fragment and a claim makes it more likely that this query fragment appears in the underlying query. Likewise, a query result that matches the claimed value in text indicates a higher probability of this query being the proper match (under the assumption that a majority of claims in a professional text document are correct). As a result, AggChecker calculates claim-specific probability distributions over queries as follows (based on Bayes rule):

$$\Pr(Q_c|S_c, E_c) \propto \Pr(S_c \wedge E_c|Q_c) \cdot \Pr(Q_c) \quad (1)$$

We derive prior query probabilities $\Pr(Q_c)$ from p, which incorporate the document topic into the claim-specific probability distribution. We also obtain the following by assum-

Table 1: **Examples for erroneous claims.**

Erroneous Claim	Author Comment	Ground Truth SQL Query	Correct Value
There were only four previous lifetime bans in my database - **three** were for repeated substance abuse, one was for gambling. [3]	Yes – the data was updated on Sept. 22, and the article was originally published on Aug. 28. There's a note at the end of the article, but you're right the article text should also have been updated.	SELECT COUNT(*) FROM NFLSUSPENSIONS WHERE GAMES = 'INDEF' AND CATEGORY = 'SUBSTANCE ABUSE, REPEATED OFFENSE'	4
Obama has spoken widely, in **17** states and D.C., while giving commencement addresses no more than three times in any one state (New York). [4]	... the likely explanation is that the data on GitHub does not include speeches Obama gave in 2016.	SELECT COUNTDISTINCT(STATE) FROM PRESIDENTIALSPEECHES WHERE PRESIDENT_NAME = 'BARACK OBAMA'	16
This year, **56,033** coders in 173 countries answered the call. [12]	Our analyst's recollection is that three of the responses were from people who "stumbled" into the survey and had no business taking it, so they were excluded from the data file and the reporting... but meanwhile, the team building the website were writing copy based on the initial count of respondents.	SELECT COUNT(*) FROM STACKOVERFLOW2016	56,030

ing independence between relevance scores and evaluation results:

$$\Pr(S_c \wedge E_c|Q_c) = \Pr(S_c|Q_c) \cdot \Pr(E_c|Q_c) \quad (2)$$

Our probabilistic model relies on a fundamental property of text documents: there is usually a primary focus in every text summary. If we know the document-specific theme (p), it helps the system to recognize SQL queries behind claims (Q_c). Similarly, if we correctly identify the SQL queries associated with the claims, the system can infer the main focus of the text. This circular dependency motivates an EM approach. AggChecker iteratively alternates between an expectation step (inferring query candidates using document priors) and a maximization step (inferring document priors from query candidates).

2.3 Query Evaluation

AggChecker exploits efficient query evaluation strategies to increase the translation accuracy of claims into queries. In fact-checking, we are not only given a natural language claim but also a claimed result. If a query result matches the claimed value, it is a strong evidence for the query candidate being the underlying query behind that claim. Hence, we can partially overcome the difficulty in natural language understanding by evaluating alternative query candidates at a massive scale.

For each claim, the system efficiently evaluates a large number of query candidates (i.e., aggregate queries) by merging them into a single CUBE query. Note that these query candidates are similar since the probabilistic model generates them based on the same input of keyword-based relevance scores. In addition, query candidates of different claims might also overlap because these claims adhere to a common theme of a text document. Thus, we cache the query results from previous claims to avoid redundant query evaluations for the current claim. Whenever we can gain by reusing the cached values, the system rewrites the cube query by removing previously processed query candidates.

Lastly, we exploit the property that we are only interested in finding out whether the query result matches the claimed value. Our system can skip the computations of query candidates that cannot yield a matching result. This is achieved by an incremental evaluation strategy where we start by evaluating a cube of smallest extent and then gradually increase the dimension of evaluated cubes. For instance, if a COUNT aggregate with no restriction produces a result lower than the claimed value, there is no point in executing query candidates with more restrictions which only results in smaller or equal values. On the other hand, incremental evaluation introduces additional overheads like reading the data multiple times. Since the incremental approach depends on the data properties, our system relies on a cost model to automatically choose for each data set whether to activate this strategy or not.

2.4 User Interface

AggChecker shows claim verification results to users via an interactive user interface. For each claim, the system provides a natural language description of the most likely query translation. If necessary, users may make corrections to a query translation giving feedback to the system. More details are presented in Section 3.2.

3. DEMONSTRATION

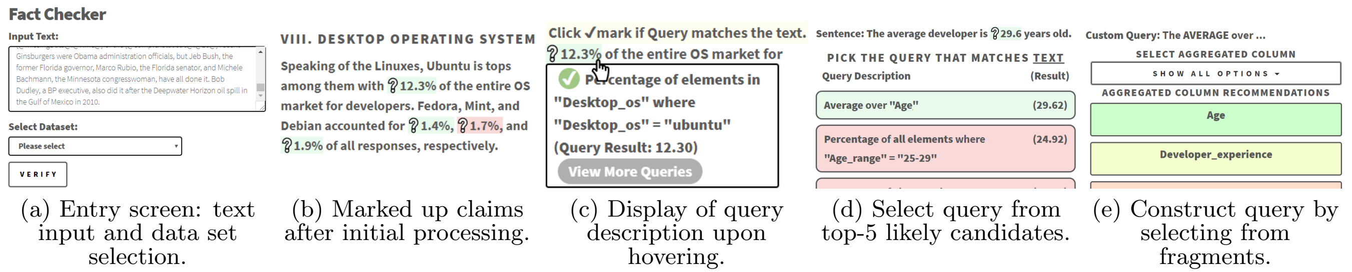
We present AggChecker as an online web application service which can support many users simultaneously. We show how our system can be used to fact-check text summaries of relational data using a variety of real-world data sets.

3.1 Real-World Data Set

We uploaded 53 real-world articles (with a total of 392 claims) and their associated data sets such that the audience can easily try out our system without the inconvenience of manually entering the text input. This includes newspaper articles from the New York Times [14], FiveThirtyEight [1], Vox [15], summaries of developer surveys on Stack Overflow [11], and Wikipedia [16] articles. In fully automated mode, AggChecker has an F1 score of 47.9% in identifying erroneous claims (when evaluated on this data set) [6]. Also, for 58.4% and 68.9% of the claims, the correct query translation is within the top-1 and top-10 likely query translations proposed by our system. Table 1 presents a subset of erroneous claims AggChecker found in the 53 articles. Furthermore, we contacted the authors of the articles to verify that these claims are truly erroneous (see the column "Author Comment").

3.2 Usage Tutorial

We walk through a usage scenario of AggChecker to describe its user interface and functionalities. A typical use



(a) Entry screen: text input and data set selection. (b) Marked up claims after initial processing. (c) Display of query description upon hovering. (d) Select query from top-5 likely candidates. (e) Construct query by selecting from fragments.

Figure 2: Screenshots from the AggChecker user interface.

case would be data journalists using our tool to double-check claims in their article before a final submission.

The first step in using AggChecker is to load the raw relational data into the database. In our demonstration, the data will be already loaded into the database, but our online web application also provides the functionality to upload new data as a comma-separated values (CSV) file.

Next step is to enter a natural language text that users want to fact-check and select the corresponding data set as in Figure 2(a). In our demonstration, we select one of the articles and click on the "Verify" button to initiate the process described in Section 2. After the initial process, the system shows the input text with two-color markup where light green and light red indicate tentatively correct and incorrect claims, respectively (see Figure 2(b)).

AggChecker shows a pop-up with the natural language description of the most likely SQL query when we hover over a marked-up claim as in Figure 2(c). If this query matches the claim, we simply click on the check mark located in the top-left corner of the pop-up. Although AggChecker identifies the right SQL query in more than 50% of the claims according to our experimental results, the system also provides alternative means to explore more possibilities when this is not the case.

The first alternative is to click on the "View More Queries" button which presents a list of top-5 likely query candidates from the probabilistic model (see Figure 2(d)). Their query results are also shown to guide the user in selecting the appropriate SQL query. We click on one of the queries if it matches the claim. When none of the candidates fits the description of the claim, we click on the "View Even More Queries" button to see five more likely query candidates.

The second alternative is to construct a query from scratch using query fragments recommended by AggChecker. The "Build Query" button shows a pop-up where we can select an aggregation function, an aggregated column, and restrictions to formulate an SQL query that matches the semantics of a claim. For each query fragment, our system ranks possible candidates by their probabilities and illustrates corresponding background colors as in Figure 2(e).

We iterate these steps until all claims are accurately associated with their underlying SQL queries. By this demonstration, the audience can learn how to use the interactive features of AggChecker.

3.3 Additional Analysis.

In addition, our online web application allows users to choose the query processing strategies themselves. Users can experience the performance improvements as the system gradually exploits efficient processing strategies. Moreover, our system presents detailed analysis of each article including the execution time per system components, the final priors of SQL query fragments, and the ratio of cached query candidates against the executed ones.

4. REFERENCES

[1] ESPN Inc. FiveThirtyEight. `http://fivethirtyeight.com/`.

[2] C. Fellbaum and G. Miller. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[3] FiveThirtyEight. The NFL's Uneven History Of Punishing Domestic Violence, 2014.

[4] FiveThirtyEight. Sitting Presidents Give Way More Commencement Speeches Than They Used To, 2016.

[5] N. Hassan, G. Zhang, F. Arslan, J. Caraballo, D. Jimenez, S. Gawsane, S. Hasan, M. Joseph, A. Kulkarni, A. K. Nayak, V. Sable, C. Li, and M. Tremayne. ClaimBuster: The First-ever End-to-end Fact-checking System. *PVLDB*, 10(12):1945–1948, 2017.

[6] S. Jo, I. Trummer, W. Yu, X. Wang, C. Yu, D. Liu, and N. Mehta. Verifying Text Summaries of Relational Data Sets. In *SIGMOD*, pages 299–316, 2019.

[7] F. Li and H. V. Jagadish. NaLIR: an interactive natural language interface for querying relational databases. In *SIGMOD*, pages 709–712, 2014.

[8] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL*, pages 55–60, 2014.

[9] G. A. Miller. WordNet: A Lexical Database for English. *Commun. ACM*, 38(11):39–41, 1995.

[10] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Ozcan. ATHENA: An ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.

[11] Stack Exchange, Inc. Stack Overflow Insights. `https://insights.stackoverflow.com/survey/`.

[12] Stack Exchange, Inc. Developer Survey Results 2016, 2016.

[13] The Apache Software Foundation. Apache Lucene Core. `https://lucene.apache.org/core/`, 2017.

[14] The New York Times Company. The Upshot. `https://www.nytimes.com/section/upshot/`.

[15] Vox Media. Vox. `https://www.vox.com/`.

[16] Wikipedia contributors. Wikipedia, The Free Encyclopedia. `https://en.wikipedia.org/`.