Join Optimization of Information Extraction Output: Quality Matters!

Alpa Jain[1], Panagiotis G. Ipeirotis[2], AnHai Doan[3], Luis Gravano[1]
[1]Columbia University, [2]New York University, [3]University of Wisconsin-Madison

***Abstract*— Information extraction (IE) systems are trained to extract specific relations from text databases. Real-world applications often require that the output of multiple IE systems be joined to produce the data of interest. To optimize the execution of a join of multiple extracted relations, it is not sufficient to consider only execution time. In fact, the quality of the join output is of critical importance: unlike in the relational world, different join execution plans can produce join results of widely different quality whenever IE systems are involved. In this paper, we develop a principled approach to understand, estimate, and incorporate output quality into the join optimization process over extracted relations. We argue that the output quality is affected by (a) the configuration of the IE systems used to process documents, (b) the document retrieval strategies used to retrieve documents, and (c) the actual join algorithm used. Our analysis considers several alternatives for these factors, and predicts the output quality—and, of course, the execution time—of the alternate execution plans. We establish the accuracy of our analytical models, as well as study the effectiveness of a quality-aware join optimizer, with a large-scale experimental evaluation over real-world text collections and state-of-the-art IE systems.**

I. Introduction

Many unstructured text documents contain valuable data that can be represented in structured form. Information extraction (IE) systems automatically extract and build structured relations from text documents, enabling the efficient use of such data in relational database systems. Real-world IE systems and architectures, such as Avatar[1], DBLife[2], and UIMA [7], view IE systems as blackboxes and "stitch" together the output from multiple such blackboxes to produce the data of interest. A common operation at the heart of these multi-blackbox systems is thus *joining* the output from the IE systems. Accordingly, recent work [7], [11], [17] has started to study this important problem of *processing joins over multiple IE systems*.

Just as in traditional relational join optimization, efficiency is, of course, important when joining the output of multiple IE systems. Existing work [7], [17] has thus focused on this aspect of the problem, which is critical because IE can be time-consuming (e.g., it often involves expensive text processing operations such as part-of-speech and named-entity tagging). Unlike in the relational world, however, the *join output quality* is of critical importance, because different join execution plans might differ drastically in the quality of their output. Several factors influence the output quality, as we discuss below. The following example highlights one such factor, namely, how errors by individual IE systems impact the join output quality.

[1]http://www.almaden.ibm.com/cs/projects/avatar
[2]http://www.dblife.cs.wisc.edu

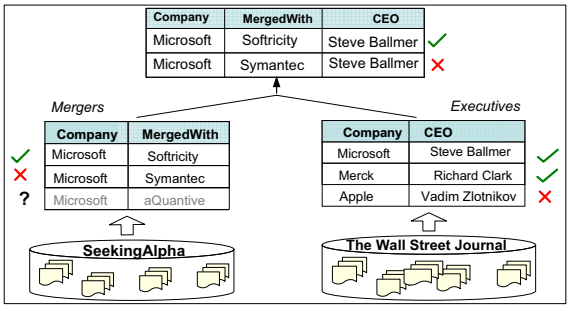


Fig. 1. Joining information derived from multiple extraction systems.

Example 1: Consider two text databases, the blog entries from SeekingAlpha (SA), a highly visible blog that discusses financial topics, and the archive of The Wall Street Journal (WSJ) newspaper. These databases embed information that can be used to answer a financial analyst's query (e.g., expressed in SQL) asking for all companies that recently merged, including information on their CEOs (see Figure 1). To answer such a query, we can use IE systems to extract a *Mergers*⟨*Company, MergedWith*⟩ relation from SA and an *Executives*⟨*Company, CEO*⟩ relation from WSJ. For *Mergers*, we extract tuples such as ⟨Microsoft, Softricity⟩, indicating that Microsoft merged with Softricity; for *Executives*, we extract tuples such as ⟨Microsoft, Steve Ballmer⟩, indicating that Steve Ballmer has been a CEO of Microsoft. After joining all the extracted tuples, we can construct the information sought by the analyst. Unfortunately, the join result is far from perfect. As shown in Figure 1, the IE system for *Mergers* incorrectly extracted tuple ⟨Microsoft, Symantec⟩, and failed to extract tuple ⟨Microsoft, aQuantive⟩. Missing or erroneous tuples, in turn, hurt the quality of join results. For example, the erroneous tuple ⟨Microsoft, Symantec⟩ is joined with the correct tuple ⟨Microsoft, Steve Ballmer⟩ from the *Executives* relation to produce an erroneous join tuple ⟨Microsoft, Symantec, Steve Ballmer⟩. □

A key observation that we make in this paper is that different join execution plans for extracted relations can *differ vastly in their output quality*. Therefore, considering the expected output quality of each candidate plan is of critical importance, and is at the core of this paper. The output quality of a join execution plan depends on (a) the configuration and characteristics of the IE systems used by the plan to process the text documents, and (b) the document retrieval strategies used by the plan to retrieve the documents for extraction. Previous work has recognized the importance of output quality for single relations [10], [8]. Recent work [11] has also considered these two factors for choosing a join execution plan over multiple extracted relations.

Unfortunately, earlier work has failed to consider a third, important factor, namely, (c) the *choice of join algorithm*. In this paper, we introduce and analyze three fundamentally different *join execution algorithms* for information extraction tasks, which differ in the extent of interaction between the extraction of the relations (e.g., from independent extraction to a completely interleaved extraction), or in the way we retrieve and process database documents (e.g., scan- or query-based retrieval). Our analysis reveals that the choice of join algorithm plays a crucial role in determining the overall output quality of a join execution plan, just as this choice crucially affects execution time in a relational model setting. We will see that even a simple two-way join has a vast execution plan space, with each execution plan exhibiting unique output quality and execution efficiency characteristics.

Understanding how join algorithms work, in concert with other factors such as the choice of extraction systems and their configurations, and the choice of document retrieval strategies, is thus crucial to optimize the processing of a join query. During optimization, we need to answer challenging questions: How should we configure the underlying IE systems? What is the correct balance between precision and recall for the IE systems? Should we retrieve and process all the database documents, or should we selectively retrieve and process only a small subset? What join execution algorithm should we use, and what is the impact of this choice on the output quality? To answer these questions, we derive equations for the execution efficiency and output quality of a join execution plan as a function of the choice of IE systems and their configurations, the choice of document retrieval strategies, and the choice of join algorithm. To the best of our knowledge, this paper presents the first holistic, in-depth study—incorporating all the above factors—of the output quality of join execution plans.

A substantial challenge that we also address is defining and extracting appropriate, comprehensive database statistics to guide the join optimization process in a quality-aware manner. As a key contribution of this paper, we show how to build rigorous statistical inference techniques to estimate the parameters necessary for our analytical models of output quality; furthermore, our parameter estimation happens efficiently, *on-the-fly* during the join execution. As another key contribution, we develop an end-to-end, quality-aware join optimizer that adaptively changes join execution strategies if the available statistics suggest that a change is desirable.

In summary, the main contributions of this paper are:

- We introduce a principled approach to estimate the output quality of a join execution and incorporate quality into the join optimization process over multiple extracted relations.
- We present an end-to-end, quality-aware join optimization approach based on our analytical models, as well as effective methods to estimate all necessary model parameters.
- We establish the accuracy of our output quality models and the effectiveness of our join optimization approach through an extensive experimental evaluation over real-life text collections and state-of-the-art IE systems.

II. Related Work

Information extraction from text has received much attention in the database, AI, Web, and KDD communities (see [3], [6] for tutorials). The majority of the efforts have considered the construction of *a single extraction system* that is as accurate as possible (e.g., using HMMs and CRFs [3], [14]). Approaches to improve the efficiency of the IE process have developed specialized document retrieval strategies; one such approach is the QXtract system [2], which uses machine learning to derive keyword queries that identify documents rich in target information. We use QXtract in our work.

Our earlier work [10], [12] studied the task of extracting just one relation, not our *join* problem. Specifically, in [10] we studied the document retrieval strategies considered in this paper for the goal of efficiently achieving a desired *recall* for a single-relation extraction task. The analysis in [10] assumes a *perfect* IE system (i.e., all generated tuples are good). On the other hand, in [12] we studied document retrieval strategies for single-relation extraction while accounting for extraction imprecision. Our current work builds upon the statistical models presented in [10], [12], extending them for multiple IE systems.

Real-world applications often require *multiple IE systems* [6], [17]. Hence, the problem of developing and optimizing IE programs that consist of multiple IE systems has received growing attention [16]. Some of the existing solutions write such programs as declaratively as possible (e.g., UIMA [7], GATE [5], Xlog [17]), while considering only the execution time in their analysis.

In prior work [11], we presented a query optimization approach for simple SQL queries involving joins while accounting for both execution time and output quality. Our earlier paper considered only *one* simple heuristic to estimate the quality of *one* simple join algorithm, namely, the IDJN algorithm, discussed and analyzed in this paper (Section IV). Our current work substantially expands on [11] by modeling an extended family of join algorithms and showing how to pick the best option dynamically. To the best of our knowledge, our current work is the first to carry out an in-depth output quality analysis of a variety of join execution plans over multiple IE systems.

III. Understanding Join Quality

In this section, we provide background on the problem of joining relations extracted from text databases. We discuss important aspects of the problem that affect the overall *quality* of the join results. We define the family of join execution plans that we consider, as well as user-specified quality preferences.

A. Tuning Extraction Systems: Impact on Extraction Quality

Extraction is a noisy process, and the extracted relations may contain erroneous tuples or miss valid tuples. An extracted relation can be regarded as consisting of *good* tuples, which are the correctly extracted tuples, and *bad* tuples, which are the erroneous tuples. For instance, in Figure 1, *Mergers* consists of one good tuple, ⟨*Microsoft, Softricity*⟩, and one bad tuple, ⟨*Microsoft, Symantec*⟩. To control the quality of such extracted relations, IE systems often expose multiple tunable "knobs"

that affect the proportion of *good* and *bad* tuples in the IE output. These knobs may be decision thresholds, such as the minimum confidence required before generating a tuple from text. We denote a particular configuration of such IE-specific knobs by θ. In our earlier work [15], we showed how we can robustly characterize the effect of such knob settings for an individual IE system, which we briefly review next. Specifically, given a knob configuration θ for an IE system, we can capture the effect of θ over a database D using two values: (a) the *true positive rate* $tp(\theta)$ is the fraction of good tuples that appear in the IE output over all the good tuples that could be extracted from D with the IE system across all possible knob configurations, while (b) the *false positive rate* $fp(\theta)$ is the fraction of bad tuples that appear in the IE output over all the bad tuples that could be extracted from D with the IE system across all possible knob configurations. To define $tp(\theta)$ and $fp(\theta)$ we need to know the sets of *all* possible good and bad tuples, which serve as normalizing factors for $tp(\theta)$ and $fp(\theta)$, respectively. In practice, we estimate $tp(\theta)$ and $fp(\theta)$ using a development set of documents and "ground truth" tuples [12].

B. Choosing Document Retrieval Strategies: Impact on Extraction Quality

Analogous to the above classification of the tuples extracted by an IE system E from a database D, we can classify each document in D with respect to E as a *good* document, if E can extract at least one good tuple from the document, as a *bad* document, if E can extract only bad tuples from the document, or as an *empty* document, if E cannot extract any tuples—good or bad—from the document. Ideally, when processing a text database with an IE system, we should focus on *good* documents and process as few *empty* documents as possible, for efficiency reasons; we should also process as few *bad* documents as possible, for both efficiency and output quality reasons. To this end, several document retrieval strategies have been introduced in the literature [10], [11]:

Scan (*SC*) sequentially retrieves and processes each document in a database. While this strategy is guaranteed to process all *good* documents, it also processes all the *empty* and *bad* ones, and may then introduce many bad tuples.

Filtered Scan (*FS*) is another scan-based strategy; instead of processing all available documents, *FS* uses a document classifier to decide whether a document is *good* or not. *FS* avoids processing all documents, and thus tends to have fewer bad tuples in the output. However, since the classifier may also erroneously reject *good* documents, *FS* might not include all the good tuples in the output.

Automatic Query Generation (*AQG*) is a query-based strategy that issues (automatically generated [2]) queries to the database that are expected to retrieve *good* documents. For example, *AQG* may derive—using machine learning—the query ["executive" AND "announced"] to retrieve documents for the *Executives* relation (see Example 1). This strategy avoids processing all the documents, but might not include all the good tuples in the output.

We now show that we can leverage these single-relation document retrieval strategies to develop *join* execution plans involving multiple extracted relations.

C. Choosing Join Execution Plans: User Preferences and Impact on Extraction Quality

In this paper, we focus on *binary natural joins*, involving two extracted relations; we leave higher order joins as future work. As discussed above, and unlike in the relational world, different join execution plans in our text-based scenario can differ not only in their execution time, but also in the quality of the join results that they produce. The output quality and, of course, the execution time is affected by (a) the configuration of the IE systems used to process the database documents, as argued in Section III-A, and (b) the document retrieval strategies used to select the documents for processing, as argued in Section III-B. Interestingly, (c) the choice of join algorithms also has an impact on the output quality and execution time, as we will see. We thus define a *join execution plan* as follows:

Definition 3.1: **[Join Execution Plan]** *Consider two databases D_1 and D_2, as well as two IE systems E_1 and E_2. Assume E_i is trained to extract relation R_i from D_i ($i = 1, 2$). A* join execution plan *for computing $R_1 \bowtie R_2$ is a tuple $\langle E_1\langle\theta_1\rangle, E_2\langle\theta_2\rangle, X_1, X_2, JN\rangle$, where θ_i specifies the knob configuration of E_i (see Section III-A) and X_i specifies the document retrieval strategy for E_i over D_i (see Section III-B), for $i = 1, 2$, while JN is the choice of join algorithm for the execution, as we will discuss below.* □

Given a join execution plan S over databases D_1 and D_2, the *execution time* $Time(S, D_1, D_2)$ is the total time required for S to generate the join results from D_1 and D_2. We identify the important components of execution time for different join execution plans in Section V, where we will also analyze the output quality associated with each plan.

We now introduce some additional notation that will be needed in our output-quality analysis in the remainder of the paper. Recall from Section III-A that the tuples that an IE system extracts for a relation can be classified as *good* tuples or *bad* tuples. Analogously, we can also classify the *attribute value occurrences* in an extracted relation according to the tuples where these values occur. Specifically, consider an attribute value a and a tuple t in which a appears. We say that the occurrence of a in t is a *good attribute value occurrence* if t is a good tuple; otherwise, this is a *bad attribute value occurrence*. Note that an attribute value might have *both* good and bad occurrences. For instance, in Figure 1 the value "*Microsoft*" has both a good occurrence in (good) tuple ⟨*Microsoft, Softricity*⟩ and a bad occurrence in (bad) tuple ⟨*Microsoft, Symantec*⟩. We denote the set of good attribute value occurrences for an extracted relation R_i by Ag_i and the set of bad attribute value occurrences by Ab_i.

Consider now a join $R_1 \bowtie R_2$ over two extracted relations R_1 and R_2. Just as in the single-relation case, the join $T_{R_1 \bowtie R_2}$ contains *good* and *bad* tuples, denoted as $\mathbf{Tgood}_{\bowtie}$ and $\mathbf{Tbad}_{\bowtie}$, respectively. The tuples in $\mathbf{Tgood}_{\bowtie}$ are the result of joining only good tuples from the base relations; all

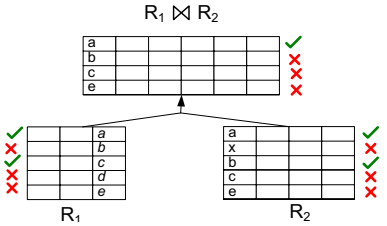


Fig. 2. Composition of $R_1 \bowtie R_2$ from extracted relations R_1 and R_2.

other combinations result in bad tuples. Figure 2 illustrates this point using example relations R_1 and R_2, with 2 good and 3 bad tuples each. In this figure, we have $Ag_1 = \{a, c\}$ and $Ab_1 = \{b, d, e\}$ for relation R_1, and $Ag_2 = \{a, b\}$ and $Ab_2 = \{x, c, e\}$ for relation R_2. The composition of the join tuples yields $|\mathbf{Tgood}_{\bowtie}| = 1$ and $|\mathbf{Tbad}_{\bowtie}| = 3$.

User Preferences: In our IE-based scenario, there is a natural trade-off between output quality and execution efficiency. Some join execution plans might produce "quick-and-dirty" results, while other plans might result in high-quality results that require a long time to produce. Ultimately, the right balance between quality and efficiency is user-specific, so our query model includes such user preferences as an important feature. One approach for handling these user preferences, which we follow in this paper, is for users to specify the desired output quality—which should be reached as efficiently as possible—in terms of the minimum number τ_g of good tuples that they request, together with the maximum number τ_b of bad tuples that they can tolerate, so that $|\mathbf{Tgood}_{\bowtie}| \geq \tau_g$ and $|\mathbf{Tbad}_{\bowtie}| \leq \tau_b$.[3] Other cost functions can be designed on top of this lower level model: examples include minimum precision at top-k results, minimum recall at the end of the execution, or a goal to maximize a weighted combination of precision and recall within a pre-specified execution time budget. For conciseness and clarity, in our work the user quality requirements are expressed in terms of τ_g and τ_b, as discussed above.

IV. Join Algorithms for Extracted Relations

As argued above, the choice of join algorithm is one of the key factors affecting the result quality. We now briefly discuss three alternate join algorithms, which we later analyze in terms of their output quality and execution efficiency. Following Section III-C, each join algorithm will attempt to meet the user-specified quality requirements as efficiently as possible. This goal is then related to that of *ripple joins* [9] for online aggregation, which minimize the time to reach user-specified performance requirements. Our discussion of the alternate join algorithms builds on the ripple join principles.

As we will see, the join algorithms base their stopping conditions on the user-specified quality requirements, given as τ_g and τ_b bounds on the number of good and bad tuples in the join output. Needless to say, the join algorithms do not have any a-priori knowledge of the correctness of the extracted tuples, so the algorithms will rely on estimates to decide when the quality requirements have been met (see Section V). Also, during execution a join algorithm might

[3] A natural alternative formulation is to specify the desired output quality as percentages of the total number of tuples [10].

Input: number of good tuples τ_g, number of bad tuples τ_b, $E_1\langle\theta_1\rangle$, $E_2\langle\theta_2\rangle$
Output: $R_1 \bowtie R_2$
$\mathbf{R}_j = \emptyset$ /* tuples produced for $R_1 \bowtie R_2$ */
$Tr_1 = \emptyset$, $Tr_2 = \emptyset$ /* tuples extracted for R_1 and R_2 */
$Dr_1 = \emptyset$, $Dr_2 = \emptyset$ /* documents retrieved from D_1 and D_2 */
while {*estimated # good tuples in* $\mathbf{R}_j < \tau_g$} *AND* {*estimated # bad tuples in* $\mathbf{R}_j \leq \tau_b$} **do**
  if $|Dr_1| < |D_1|$ **then**
    Retrieve an unseen document d from D_1 and add to Dr_1
    Process d using E_1 at θ_1 to generate tuples t_1
  end
  if $|Dr_2| < |D_2|$ **then**
    Retrieve an unseen document d from D_2 and add to Dr_2
    Process d using E_2 at θ_2 to generate tuples t_2
  end
  $T_{join} = (t_1 \bowtie t_2) \cup (Tr_1 \bowtie t_2) \cup (t_1 \bowtie Tr_2)$
  $Tr_1 = Tr_1 \cup t_1$
  $Tr_2 = Tr_2 \cup t_2$
  $\mathbf{R}_j = \mathbf{R}_j \cup T_{join}$
  if $\{|Dr_1| = |D_1|\}$ *AND* $\{|Dr_2| = |D_2|\}$ **then**
    return $\mathbf{R}_j$
  end
end
return $\mathbf{R}_j$

Fig. 3. The IDJN algorithm using *Scan*.

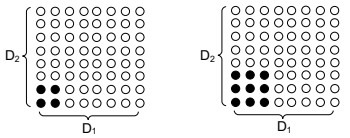


Fig. 4. Exploring $D_1 \times D_2$ with IDJN using *Scan*.

estimate that the quality requirements cannot be met, in which case the join optimizer may build on the current execution with a different join execution plan or, alternatively, discard any produced results and start a new execution plan from scratch (see Section VI).

A. Independent Join

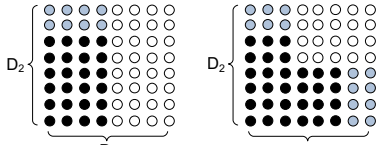
The Independent Join algorithm (IDJN) [11] computes a binary join by extracting the two relations *independently* and then joining them to produce the final output. To extract each relation, IDJN retrieves the database documents by choosing from the document retrieval strategies in Section III-A, namely, *Scan*, *Filtered Scan*, and *Automatic Query Generation*.

Figure 3 describes the IDJN algorithm for the settings of Definition 3.1 and for the case where the document retrieval strategy is *Scan*. IDJN receives as input the user-specified output quality requirements τ_g and τ_b (see Section III), and the relevant extraction systems $E_1\langle\theta_1\rangle$ and $E_2\langle\theta_2\rangle$. IDJN sequentially retrieves documents for both relations, runs the extraction systems over them, and joins the newly extracted tuples with all the tuples from previously seen documents. Conceptually, the join algorithm can be viewed as "traversing" the Cartesian product $D_1 \times D_2$ of the database documents, as illustrated in Figure 4: the horizontal axis represents the documents in D_1, the vertical axis represents the documents in D_2, and each element in the grid represents a document pair in $D_1 \times D_2$, with a dark circle indicating an already visited element. (The documents are displayed in the order of retrieval.) Figure 4 shows a "square" version of IDJN, which retrieves documents from D_1 and D_2 at the same rate; we can generalize this algorithm to a "rectangle" version that retrieves documents from the databases at different rates.

The number of documents to explore will depend on the user-specified values for τ_g and τ_b, and also on the choice of the retrieval strategies for each relation. When using *Filtered Scan*, we may filter out a retrieved document from a database and, as a result, some portion of $D_1 \times D_2$ will remain

Input: number of good tuples τ_g, number of bad tuples τ_b, $E_1\langle\theta_1\rangle$, $E_2\langle\theta_2\rangle$
Output: $R_1 \bowtie R_2$
$\mathbf{R}_j = \emptyset$ /* tuples produced for $R_1 \bowtie R_2$ */
/* R_1 is the outer relation and R_2 is the inner relation */
$Tr_1 = \emptyset$, $Tr_2 = \emptyset$ /* tuples extracted for R_1 and R_2 */
$Dr_1 = \emptyset$, $Dr_2 = \emptyset$ /* documents retrieved from D_1 and D_2 */
while {*estimated # good tuples in* $\mathbf{R}_j < \tau_g$} *AND* {*estimated # bad tuples in* $\mathbf{R}_j \leq \tau_b$} **do**
  $Q_s = \emptyset$ /* queries for the inner relation */
  Retrieve an unseen document d from D_1 and add to Dr_1
  Process d using E_1 at θ_1 to generate tuples t_1
  Generate keyword queries from t_1 and add to Q_s
  $Tr_1 = Tr_1 \cup t_1$
  $\mathbf{R}_j = \mathbf{R}_j \cup (t_1 \bowtie Tr_2)$
  foreach *query* $q \in Q_s$ **do**
    Issue q to D_2 to retrieve unseen matching documents and add them to Dr_2
    Process all unprocessed documents in Dr_2 using E_2 at θ_2 to generate tuples t_2
    $Tr_2 = Tr_2 \cup t_2$
    $\mathbf{R}_j = \mathbf{R}_j \cup (Tr_1 \bowtie t_2)$
  end
  if $|Dr_1| = |D_1|$ **then**
    return $\mathbf{R_j}$
  end
end
return $\mathbf{R_j}$

Fig. 5. The OIJN algorithm using *Scan* for the outer relation.



(a) (b)

Fig. 6. Exploring $D_1 \times D_2$ with (a) OIJN and (b) ZGJN.

unexplored. Similarly, when using *Automatic Query Generation*, the maximum number of documents retrieved from a database may be limited, resulting in a similar effect.

B. Outer/Inner Join

The IDJN algorithm does not effectively exploit any existing keyword-based indexes on the text databases. Existing indexes can be used to guide the join execution towards processing documents likely to contain a joining tuple. The next join algorithm, Outer/Inner Join (OIJN), shown in Figure 5, corresponds to a nested-loop join algorithm in the relational world. OIJN thus picks one of the relations as the "outer" relation and the other as the "inner" relation. (Our analysis in Section V can be used to identify which relation should serve as the outer relation in a join execution.) OIJN retrieves documents for the outer relation using one of the document retrieval strategies and processes them using an appropriate extraction system. OIJN then generates keyword queries using the values for the joining attributes in the extracted outer relation. Using these queries, OIJN retrieves and processes documents for the inner relation that are likely to contain the "counterpart" for the already seen outer-relation tuples.

Figure 6(a) illustrates the traversal through $D_1 \times D_2$ for the OIJN algorithm: each querying step corresponds to a *complete* probe of the inner relation's database, which sweeps out an entire row of $D_1 \times D_2$. Thus, OIJN effectively traverses the space, biasing towards documents likely to contain joining tuples from the inner relation, which may result in an efficient refinement over IDJN. However, the maximum number of documents that can be retrieved via a query may be limited by the search interface, which, in turn, limits the maximum number of tuples retrieved using OIJN. The impact of this limit on the number of matching documents is denoted in Figure 6(a) as gray circles that depict some unexplored area

Input: number of good tuples τ_g, number of bad tuples τ_b, $E_1\langle\theta_1\rangle$ and $E_2\langle\theta_2\rangle$, Q_{seed}
Output: $R_1 \bowtie R_2$
$\mathbf{R}_j = \emptyset$ /* tuples produced for $R_1 \bowtie R_2$ */
$Tr_1 = \emptyset$, $Tr_2 = \emptyset$ /* tuples extracted from R_1 and R_2 */
$Dr_1 = \emptyset$, $Dr_2 = \emptyset$ /* documents retrieved from D_1 and D_2*/
$Q_1 = Q_{seed}$, $Q_2 = \emptyset$ /* queries issued to D_1 and D_2 */
while $Q_1 \neq \emptyset$ *OR* $Q_2 \neq \emptyset$ **do**
  if $Q_1 \neq \emptyset$ **then**
    Pull next query q_1 out of Q_1
    Issue q_1 to D_1 to retrieve unseen matching documents and add them to Dr_1
    Process all unprocessed documents in Dr_1 using E_1 at θ_1 to generate tuples t_1
    Generate keyword queries from t_1 and append to Q_2
    $Tr_1 = Tr_1 \cup t_1$
    $\mathbf{R}_j = \mathbf{R}_j \cup (t_1 \bowtie Tr_2)$
  end
  if {*estimated # good tuples in* $\mathbf{R}_j \geq \tau_g$} OR {*estimated # bad tuples in* $\mathbf{R}_j > \tau_b$} **then**
    return $\mathbf{R}_j$
  end
  if $Q_2 \neq \emptyset$ **then**
    Pull next query q_2 out of Q_2
    Issue q_2 to D_2 to retrieve unseen matching documents and add them to Dr_2
    Process all unprocessed documents in Dr_2 using E_2 at θ_2 to generate tuples t_2
    Generate keyword queries from t_2 and append to Q_1
    $Tr_2 = Tr_2 \cup t_2$
    $\mathbf{R}_j = \mathbf{R}_j \cup (Tr_1 \bowtie t_2)$
  end
  if {*estimated # good tuples in* $\mathbf{R}_j \geq \tau_g$} OR {*estimated # bad tuples in* $\mathbf{R}_j > \tau_b$} **then**
    return $\mathbf{R}_j$
  end
end

Fig. 7. The ZGJN algorithm.

in the $D_1 \times D_2$ space.

C. Zig-Zag Join

The Zig-Zag Join (ZGJN) algorithm generalizes the idea of using keyword queries from OIJN, so that we can now query for *both* relations and interleave the extraction of the two relations; see Figure 7. Starting with one tuple extracted for one relation, ZGJN retrieves documents—via keyword querying on the join attribute values—for extracting the second relation. In turn, the tuples from the second relation are used to build keyword queries to retrieve documents for the first relation, and the process iterates, effectively alternating the role of the "outer" relation of a nested-loop execution over the two relations. Conceptually, each step corresponds to a sweep of an entire row or column of $D_1 \times D_2$, as shown in Figure 6(b). Similarly to OIJN, ZGJN can efficiently traverse the $D_1 \times D_2$ space; however, just as for OIJN, the space covered by ZGJN is limited by the maximum number of documents returned by the underlying search interface.

V. Estimating Join Quality

Each join execution plan (Definition 3.1) produces join results whose quality depends on the choice of IE system—and their tuning parameters (see Section III-A)—, document retrieval strategies (see Section III-B), and join algorithms (see Section IV). We now turn to the core of this paper, where we present *analytical models for the output quality* of the join execution plans. Specifically, for each execution plan we derive formulas for the number $|\mathbf{Tgood}_{\bowtie}|$ of good tuples and the number $|\mathbf{Tbad}_{\bowtie}|$ of bad tuples in $R_1 \bowtie R_2$ that the plan produces, as a function of the number of documents retrieved and processed by the IE systems.

A. Notation

In the rest of the discussion, we consider two text databases, D_1 and D_2, with two IE systems E_1 and E_2, where extraction system E_i extracts relation R_i from D_i ($i = 1, 2$). Table I summarizes our notation for the good, bad, and empty database documents, for the good and bad tuples and attribute values,

Symbol	Description	Symbol	Description
R_i	Extracted relations (**i = 1, 2**)		
E_i	Extraction system for R_i		
X_i	Document retrieval strategy for E_i		
D_i	Database for extracting R_i	Ag_i	Good attribute values in R_i
Dg_i	Set of good documents in D_i	Ab_i	Bad attribute values in R_i
Db_i	Set of bad documents in D_i	$g_i(a)$	Frequency of a in Dg_i
De_i	Set of empty documents in D_i	$b_i(a)$	Frequency of a in Db_i
Dr_i	Set of documents retrieved from D_i	$O_i(a)$	Frequency of a in Dr_i

TABLE I
NOTATION SUMMARY.

as well as for their frequency in the extracted relations (see Section III-C). Additionally, for a natural join attribute A[4], we denote the attribute values common to both extracted relations R_1 and R_2 as follows: $Agg = Ag_1 \cap Ag_2$, $Agb = Ag_1 \cap Ab_2$, $Abg = Ab_1 \cap Ag_2$, and $Abb = Ab_1 \cap Ab_2$. In Figure 2, $Agg = \{a\}$, $Agb = \{c\}$, $Abg = \{b\}$, and $Abb = \{e\}$.

B. Analyzing Join Execution Plans: General Scheme

We begin our analysis by sketching a general scheme to study the output of an execution plan in terms of its number of good tuples $|\mathbf{Tgood}_{\bowtie}|$ and bad tuples $|\mathbf{Tbad}_{\bowtie}|$. In later sections, we will instantiate this general scheme for the various join execution plans that we study.

Consider relations R_1 and R_2, to be extracted and joined over a single common attribute A, and let a be a value of join attribute A with $g_1(a)$ *good* occurrences in D_1[5] and $g_2(a)$ *good* occurrences in D_2. Suppose that a join execution retrieves documents Dr_1 from D_1 and documents Dr_2 from D_2, and we observe $gr_1(a)$ *good* occurrences of a in Dr_1 and $gr_2(a)$ *good* occurrences of a in Dr_2. Then, the number of good *join* tuples with $A = a$ is $gr_1(a) \cdot gr_2(a)$ (see Section III-C). Generalizing this analysis to *all* good attribute occurrences common to both relations (i.e., to the values in Agg) the total number $|\mathbf{Tgood}_{\bowtie}|$ of good tuples extracted for $R_1 \bowtie R_2$ is:

$$|\mathbf{Tgood}_{\bowtie}| = \sum_{a \in Agg} gr_1(a) \cdot gr_2(a) \quad (1)$$

where Agg is the set of join attribute values with good occurrences in both relations (see Section V-A). Among other factors, the values of $gr_1(a)$ and $gr_2(a)$ depend on the frequencies $g_1(a)$ and $g_2(a)$ of a in the complete databases D_1 and D_2. As we will see in the next sections, we can estimate the conditional expected frequency $E[gr_i(a)|g_i(a)]$ for each attribute value given the configuration of the IE systems, the choice of document retrieval strategy, and the choice of join algorithm. Assuming, for now, that we know the conditional expectations, we have:

$$E[|\mathbf{Tgood}_{\bowtie}|] = \sum_{a \in Agg} E[gr_1(a)|g_1(a)] \cdot E[gr_2(a)|g_2(a)]$$

[4]Without loss of generality, we assume a single join attribute A. We can treat the union of multiple join attributes as a "conceptual" single attribute.

[5]Conceptually, $g_i(a)$ can be defined in terms of the number of good tuples that contain attribute value a. For simplicity, we assume that each attribute value appears only once in each document. This simplification significantly reduces the complexity of the statistical model, without losing much of its accuracy, since most of the attributes indeed appear only once in each document. (We verified the latter experimentally.)

In practice, we do not know the exact frequencies $g_1(a)$ and $g_2(a)$ for each attribute value. However, we can typically estimate the probability $Pr\{g_i\}$ that an attribute value occurs g_i times in an extracted relation, using parametric or nonparametric approaches (e.g., often the frequency distribution of attribute values follows a power law [10]). So,

$$E[|\mathbf{Tgood}_{\bowtie}|] = |Agg| \cdot \sum_{g_1=1}^{|Dg_1|} \sum_{g_2=1}^{|Dg_2|} E[gr_1|g_1] \cdot E[gr_2|g_2] \cdot Pr\{g_1, g_2\}$$

The factor $Pr\{g_1, g_2\}$ is the probability that an attribute value has g_1 good occurrences in D_1 and g_2 good occurrences in D_2. We make a simplifying independence assumption so that $Pr\{g_1, g_2\} = Pr\{g_1\} \cdot Pr\{g_2\}$. Alternatively, we could assume that frequent attribute values in one relation are commonly frequent in the other relation as well, and vice versa[6]. In this scenario, we would have: $Pr\{g_1\} \approx Pr\{g_2\}$ and $Pr\{g_1, g_2\} \approx Pr\{g_1\} \approx Pr\{g_2\}$.

To compute the number $|\mathbf{Tbad}_{\bowtie}|$ of bad tuples in $R_1 \bowtie R_2$ we proceed in the same fashion, with two main differences: we need to consider three different classes of attributes, namely, Agb, Abg, and Abb, and compute the expected number of *bad* attribute occurrences in an extracted relation. Specifically,

$$|\mathbf{Tbad}_{\bowtie}| = J_{gb} + J_{bg} + J_{bb}$$

where $J_{gb} = |Agb| \cdot \sum_{g_1=1}^{|Dg_1|} \sum_{b_2=1}^{|Db_2|} E[gr_1|g_1] \cdot E[br_2|b_2] \cdot Pr\{g_1, b_2\}$. We can compute values for J_{bg} and J_{bb} using $|Abg|$ and $|Abb|$, respectively, along with appropriate tuple cardinality values.

Using this generic analysis along with the expected frequencies for the attribute occurrences, we can derive the exact composition of $R_1 \bowtie R_2$. We now complete and instantiate this analysis for the alternate join algorithms of Section IV.

C. Independent Join

Our goal is to derive the expected frequency $E[gr_i]$ for good attribute occurrences and $E[br_i]$ for bad attribute occurrences in R_i after we have retrieved Dr_i documents from D_i, given the frequencies of occurrence in D_i ($i = 1, 2$). As IDJN independently generates each relation, the analysis for $E[gr_1]$ is the same as that for $E[gr_2]$, and depends on the choice of document retrieval strategy and IE system configuration; similarly, the analysis for $E[br_1]$ is the same as that for $E[br_2]$. Hence, we ignore the relation subindex in the discussion.

We start by computing $E[gr]$ for an attribute value a with $g(a) = g$ good occurrences. We focus only on the good documents Dg in D, as good occurrences only appear in the good documents. We model a document retrieval strategy as *sampling processes over the good documents* Dg [10]. After retrieving Dgr good documents, the probability of observing k times the good attribute occurrence a in the retrieved documents follows a hypergeometric distribution, $Hyper(|Dg|, |Dgr|, g, k)$, where $Hyper(D, S, g, k) = \binom{g}{k} \cdot \binom{D-g}{S-k} / \binom{D}{S}$.

[6]This simplifying assumption may not always hold: for instance, a *Movie* attribute value that appears in a single tuple of a *Directors*⟨*Movie, Director*⟩ relation might appear in many tuples of an *Actors*⟨*Movie, Actor*⟩ relation.

We process the retrieved documents Dgr using an IE system E. As E is not perfect, even if we retrieve k documents that contain the good attribute occurrences, E examines each of the k documents independently and, with probability $tp(\theta)$ for each document, outputs the occurrence. So, we will see good occurrences in the extracted tuples only $l \leq k$ times, and l is a random variable following the binomial distribution. Thus, the expected frequency of a good attribute occurrence in an extracted relation, after processing j good documents, is:

$$E[gr||Dgr| = j] = \sum_{k=0}^{g} Hyper(|Dg|, j, g, k) \cdot \sum_{l=0}^{k} l \cdot Bnm(k, l, tp(\theta))$$

where $Bnm(n, k, p) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$ is the binomial distribution and g is the frequency of good attribute occurrences in D. The derivation for bad attribute occurrences $E[br]$ is analogous to that for $E[gr]$, but now a bad attribute occurrence can be extracted from *both* good and bad database documents.

So far, the analysis implicitly assumed that we know the exact proportion of the good and bad documents retrieved, i.e., $|Dgr|$ and $|Dbr|$. In reality, though, this proportion depends on the choice of document retrieval strategy. We analyze the effect of a document retrieval strategy next.

Scan sequentially retrieves documents for E from database D, in no specific order. Therefore, when *Scan* retrieves $|Dr|$ documents, E processes $|Dgr|$ *good* documents, where $|Dgr|$ is a random variable that follows the hypergeometric distribution. Specifically, the probability of processing exactly j good documents is $Pr(|Dgr| = j) = Hyper(|D|, |Dr|, |Dg|, j)$. We compute the probability of processing j bad documents analogously.

Filtered Scan is similar to *Scan*, except that a document classifier filters out documents that are not good candidates for containing good tuples. Thus, only documents that *survive* the classification step will be processed. Document classifiers are not perfect either, and they are usually characterized by their *true positive rate* C_{tp} and *false positive rate* C_{fp}. Intuitively, for a classifier C, the true positive rate C_{tp} is the fraction of documents in Dg classified as good, and the false positive rate C_{fp} is the fraction of documents in Db incorrectly classified as good. Therefore, the main difference from *Scan* is that now the probability of processing j *good* documents after retrieving $|Dr|$ documents from the database is:

$$Pr(|Dgr| = j) = \sum_{n=0}^{|Dr|} Hyper(|D|, |Dr|, |Dg|, n) \cdot Bnm(n, j, C_{tp})$$

We compute the probability of processing j bad documents in a similar way using C_{fp}.

Automatic Query Generation retrieves documents from D by issuing queries designed to retrieve mainly good documents. Consider the case where *AQG* has sent Q queries to the database. If a query q retrieves $g(q)$ documents and has precision $P(q)$, where $P(q)$ is the fraction of documents retrieved by q that are good, then the probability that a good document is retrieved by q is $\frac{P(q) \cdot g(q)}{|Dg|}$. Assuming that the queries sent by *AQG* are only biased towards documents in Dg, the queries are conditionally independent within Dg. In this case, the probability that a *good* document d is retrieved by at least one of the Q queries is:

$$Pr_g(d) = 1 - \prod_{i=1}^{Q} \left(1 - \frac{p(q_i) \cdot g(q_i)}{|Dg|}\right) \quad (2)$$

Since each document is retrieved independently of each other, the number of good documents retrieved (and processed) follows a binomial distribution, with $|Dg|$ trials and $Pr_g(d)$ probability of success in each trial, so $Pr(|Dgr| = j) = Bnm(|Dg|, j, Pr_g(d))$. The analysis is analogous for the bad documents.

The execution time for an IDJN execution strategy follows from the general discussion above. Consider the case when we retrieve $|Dr_1|$ documents from D_1 and $|Dr_2|$ documents from D_2 using *Scan* for both relations. In this case IDJN does not filter or query for documents, so the execution time is:

$$Time(S, D_1, D_2) = \sum_{i=1}^{2} |Dr_i| \cdot \left(t_R^i + t_E^i\right)$$

where t_R^i is the time to retrieve a document from D_i and t_E^i is the time required to process the document using the E_i IE system. For execution strategies that use *Filtered Scan* or *Automatic Query Generation*, we compute the execution time by considering the time t_F^i to filter a document, or the time t_Q^i (together with the number of queries issued $|Qs^i|$) to send and retrieve the results of a query.

D. Outer/Inner Join

For OIJN, the analysis for the outer relation is the same as that for an individual relation in IDJN: the expected frequency of (good or bad) occurrences of an attribute value depends solely on the document retrieval strategy and the IE system for the outer relation. On the other hand, the expected frequency of the attribute occurrences for the inner relation depends on the number of queries issued using attribute values from the outer relation, as well as on the IE system used to process the matching documents. Our analysis focuses on the inner relation; again, we omit the relation subindex, for simplicity.

Consider again an attribute value a with $g(a)$ good occurrences in the database, and a query q generated from a that has $H(q)$ document matches and precision $P(q)$, where $P(q)$ is the fraction of documents matching q that are good. Thus, the set of good documents that can match q is $H_g(q)$, with $|H_g(q)| = |H(q)| \cdot P(q)$. When we issue q, the subset of $H_g(q)$ documents returned is limited by the search interface. Specifically, if the search interface returns only the top-k documents for a query, for a fixed value of k, we expect to see $k \cdot P(q)$ good documents. An important observation is that the documents that match q but are not returned in the top-k results can also be retrieved by queries other than q. Thus, when we issue Qs queries and retrieve Dgr good documents, we can observe a from two disjoint sets: $k \cdot P(q)$, the good documents in the top-k answers for q, and the rest, $Dgr_{rest} = Dgr - k \cdot P(q)$.

If $Pr_q\{gr_q|g(a),q\}$ is the probability that we will observe attribute value a a total of gr_q times in the top results for q, and $Pr_r\{gr_{rest}|g(a),Dgr_{rest}\}$ is the probability that we will observe attribute value a a total of gr_{rest} times in the remainder documents, we have:

$$E[gr(a)] = \sum_{l=0}^{g(a)} l \cdot (Pr_q\{l|g(a),q\} + Pr_r\{l|g(a),Dgr_{rest}\})$$

For $Pr_q\{gr_q|g(a),q\}$, we model querying as sampling over $H_g(q)$ while drawing $k \cdot P(q)$ samples, and derive:

$$Pr_q\{gr_q|g(a),q\} = \sum_{i=0}^{g(a)} H\big(g(a),i\big) \cdot B_g(i,gr_q)$$

where $H(k,i) = Hyper(|H_g(q)|, k \cdot P(q), k, i)$ and $B_g(k,l) = Bnm(k,l,tp(\theta))$.

For $Pr_r\{gr_{rest}|g(a),Dgr_{rest}\}$, we observe that the total number $g(a)$ of documents in Dgr_{rest} is the number of documents containing a minus the good documents that matched the query. Specifically, among documents not retrieved via the query q, an attribute value can occur $g'(a)$ times, where $g'(a) = g(a) \cdot \frac{|H_g(q)|-k\cdot P(q)}{|H_g(q)|}$. We model the process of retrieving documents for a, using queries other than q, as sampling over Dg while drawing samples Dgr_{rest}, and derive:

$$Pr_r\{l|g(a),|Dgr_{rest}|=j\} = \sum_{i=0}^{g(a)} H_g\left(g'(a),i\right) \cdot B_g(i,l)$$

where $H_g(k,i) = Hyper(|Dg|,j,k,i)$. For $E[br(a)]$, i.e., a bad attribute value, we proceed similarly.

For execution time, if we retrieve Dr_o documents using *Scan* for the outer relation and send Qs queries for the inner relation, and retrieve Dr_i documents, the execution time is:

$$Time(S,D_1,D_2) = |Dr_o| \cdot (t_R^o + t_E^o) + |Dr_i| \cdot \big(t_R^i + t_E^i\big) + |Qs| \cdot t_Q^i$$

where t_R^o and t_E^o are the times to retrieve and process, respectively, a document for the outer relation, t_R^i and t_E^i are the times to retrieve and process, respectively, a document for the inner relation, and t_Q^i is the time to issue a query to the inner relation's database. The value for $|Dr_o|$ is determined so that the resulting join execution meets the user requirements.

E. Zig-Zag Join

To analyze the ZGJN algorithm, we define a *zig-zag* graph consisting of two classes of nodes: *attribute* nodes ("a" nodes) and *document* nodes ("d" nodes), and two classes of edges: *hit* edges and *generates* edges. A *hit* edge $A \rightarrow D$ connects an a node to a d node, and denotes that a generated a *hit* on d, that is, d matches the query generated using a. A *generates* edge $d \rightarrow a$ connects a d node to an a node and denotes that processing d generated a.

As an example, consider the *zig-zag* graph in Figure 8 for joining *Mergers* and *Executives* from Example 1 on the *Company* attribute. We begin with a *seed* query ["Microsoft"] for *Mergers* and issue it to the D_2 database. This query hits a document d_{21}. Processing d_{21} generates tuples for *Executives*, which contain values Microsoft and AOL for *Company*. At this

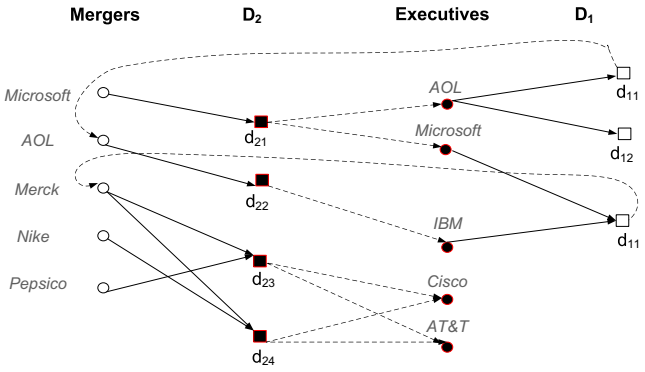


Fig. 8. Sample *zig-zag* graph for *Mergers* $\bowtie$ *Executives*.

stage, the total number of attributes generated for *Executives* is determined by the number of documents that matched the query ["Microsoft"]. Next, we issue the query ["AOL"] to D_1, which retrieves documents d_{11} and d_{12}. The total number of documents retrieved from D_1 is determined by the number of attribute values generated for *Executives* in the previous step. Processing d_{11} for *Mergers* generates a new attribute value, AOL, which is used to generate new queries for D_2, and the process continues.

The above example shows that the characteristics of a ZGJN execution are determined by the total number of attribute values and documents that could be *reached* following the edges on the *zig-zag* graph. Thus, the structure of the graph defines the execution time and the output quality for ZGJN. We study the interesting properties of a *zig-zag* graph using the theory of random graphs [15]. Specifically, we build on the single-relation approach in [10] to model our join scenario, and use generating functions to describe the properties of a *zig-zag* graph. We begin by defining two generating functions, $h_0(x)$, which describes the number of hits for a *randomly chosen* attribute value, and $ga_0(x)$, which describes the number of attributes generated from a *randomly chosen* document.

$$h_0(x) = \sum_k pa_k \cdot x^k, \qquad ga_0(x) = \sum_k pd_k \cdot x^k$$

where pa_k is the probability that a randomly chosen attribute a matches k documents, and pd_k is the probability that a randomly chosen document generates k attributes. To keep the model parameters manageable, we approximate the distribution for pa_k with the attribute frequency distribution used by our general analysis (Section V-B), as the two distributions tend to be similar. Specifically, we derive the probability that an attribute a matches k documents using the probability that a is extracted from k documents.

Our goal, however, is to study the frequency distribution of an attribute or a document chosen by *following a random edge*. For this, we use the method in [15], [10] and define functions $H(x)$ and $Ga(x)$ that, respectively, describe the attribute and the document frequency chosen by following a random edge on the *zig-zag* graph.

$$H(x) = x\frac{h_0{}'(x)}{h_0{}'(1)}, \quad Ga(x) = x\frac{ga_0{}'(x)}{ga_0{}'(1)}$$

where $h_0{}'(x)$ is the first derivative of $h_0(x)$ and $ga_0{}'(x)$ is the first derivative of $ga_0(x)$. To distinguish between the relations,

we denote the functions using subindices: $H_i(x)$ and $Ga_i(x)$, respectively, describe the attribute and the document frequency distributions for R_i ($i = 1, 2$).

We will now derive equations for the number of documents $E[|Dr_1|]$ and $E[|Dr_2|]$ retrieved from D_1 and D_2, respectively, and the number of attribute values $E[|Ar_1|]$ and $E[|Ar_2|]$ generated for relation R_1 and R_2, respectively. For our analysis, we exploit three useful properties, *Moments*, *Power*, and *Composition* of generating functions (see [15], [10]). The distribution of the total number $|Dr_2|$ of documents retrieved from D_2 using attributes from R_1 can be described by the function $Dr_2(x) = H_1(x)$. Further, the distribution of the attribute values generated from a D_2 document picked by following a random edge is given by $Ga_2(x)$. Using the *Composition* property, the distribution of the total number of attribute values generated from Dr_2 is given by the function $Ar_2(x) = H_1(Ga_2(x))$.

The total number $|Ar_2|$ of R_2 attribute values that will be used to derive the D_2 documents is a random variable with its distribution described by $Ar_2(x)$. Furthermore, the distribution of the documents retrieved by an R_2 attribute value picked by following a random edge is described by $H_2(x)$. Once again, using the *Composition* property, we describe the distribution of the total number of D_2 documents retrieved using Ar_2 attribute values using the generating function $Dr_1(x) = Ar_2(H_2(x)) = H_1(Ga_2(H_2(x)))$. To describe the total number $|Ar_1|$ of R_1 attribute values derived by processing a Dr_1 document, we compose $Dr_1(x)$ and $Ga_1(x)$, and define $Ar_1(x) = Dr_1(Ga_1(x)) = H_1(Ga_2(H_2(Ga_1(x))))$.

Next, we generalize the above functions for Q_1 queries sent from R_1 attribute values and using the *Power* property:

$$Dr_2(x) = [H_1(x)]^{|Q_1|}, \qquad Ar_2(x) = [H_1(Ga_2(x))]^{|Q_1|}$$

Finally, we compute the expected values $E[|Dr_2|]$ after we have issued Q_1 queries using R_1 attribute values. For this, we resort to the *Moments* property.

$$E[|Dr_2|] = \left[\frac{d}{dx}\left[H_1(x)\right]^{|Q_1|}\right]_{x=1}$$
$$E[|Ar_2|] = \left[\frac{d}{dx}\left[H_1(Ga_2(x))\right]^{|Q_1|}\right]_{x=1}$$

Similarly, we derive values for $E[|Dr_1|]$ and $E[|Ar_1|]$.

We derived the total number of attributes $E[|Ar_1|]$ and $E[|Ar_2|]$ for the individual relations, but we are interested in the total number of good and bad attribute occurrences generated for each relation. For this, we split the number of attributes in a relation, using the fraction of good or bad attribute occurrences in a relation. For instance,

$$E[|gr_1|] = E[|Ar_1|] \cdot \frac{|Ag_1|}{|Ag_1| + |Ab_1|}$$

Given the analysis above, we compute the execution time of a zig-zag join that satisfies the user-specified quality requirements: if we issue $|Qs^i|$ queries and retrieve $|Dr_i|$ documents for relation R_i, $i = 1, 2$, the execution time is:

$$Time(S, D_1, D_2) = \sum_{i=1}^{2} |Dr_i| \cdot \left(t_R^i + \cdot t_E^i\right) + |Qs^i| \cdot t_Q^i$$

$|Qs^1|$ and $|Qs^2|$ are the minimum values required for the output quality to meet the user-specified quality requirements.

To summarize, in this section we analyzed each join algorithm for the various choices of document retrieval strategies and IE system configurations. Our analysis resulted in formulas for the join quality composition in terms of the number of documents retrieved for each relation or the number of keyword queries issued to a database. Conversely, this analysis can be used to determine these input values for a given output quality.

VI. Incorporating Output Quality into Join Optimization

Our optimization approach applies the analysis from Section V to our general goal of selecting a join execution strategy for a given user-specified quality requirement. We now discuss how we derive various parameters used in the analysis, and present our overall optimization approach.

Estimating Model Parameters: The analysis in Section V relies on three classes of parameters, namely, the retrieval strategy-specific parameters, the database-specific parameters, and the join algorithm-specific parameters. The retrieval strategy-specific parameters are the precision $p(q_i)$ of each query q_i for *AQG*, or the classifier properties C_{tp} and C_{fp} for *FS*; the database-specific parameters are $|Dg|$, $|Db|$, $|Ag|$, $|Ab|$, as well as the document and attribute frequency distributions, for each relation, and the values for $|Agg|, |Agb|, |Abg|$, and $|Abb|$. Finally, the join-specific parameters are $H(q)$ and $P(q)$ for OIJN and ZGJN. Of these, the retrieval strategy-specific parameters and the join algorithm-specific parameters can be easily estimated in a pre-execution, offline step [10]. On the other hand, estimating the database-specific parameters is a more challenging task [12].

We estimate the parameters for each relation, separately, using *maximum likelihood estimation (MLE)* based on the approach described in [12]. Our MLE model relies on the analytical models in Section V, where we showed how to estimate the output *given the database-specific parameters*; to estimate parameters, we observe the output and *infer the database-specific parameter values* that are the most likely to generate the observed output. Due to space restrictions, we cannot present our estimation process in detail, but we provide the basic intuition behind it. Please refer to [13] for details.

While retrieving documents from database D, we observe some attributes and their frequencies in the retrieved documents. So, for an attribute a_i obtained from Dr, we use $s(a_i)$ to denote the number of documents in Dr that generated a_i. These values reveal information about the actual contents of the database. Formally, we attempt to find the database specific parameters that maximize the likelihood function:

$$\mathcal{L}(parameters) = \prod_i Pr\{s(a_i)|parameters\}$$

To find the set of parameter values that maximize $\mathcal{L}(parameters)$, we use the models from Section V to express

$Pr\{s(a)|parameters\}$ as a function of the database parameters. Using these derivations, we search the space of parameters to find the values that maximize $\mathcal{L}$. An important advantage of our estimation method is that it does not require any verification method to determine whether an observed tuple is good or bad; the estimation methods derive a probabilistic split of the observed tuples, thereby carrying out the parameter estimation process in a stand-alone fashion. We define a similar likelihood function for the document frequencies. Using the estimated parameter values for each individual relation, we then numerically derive the join-specific parameters [13].

Putting It All Together: Our optimizer takes as input the user-provided minimum number of good tuples τ_g and the maximum number of bad tuples τ_b, and picks an execution strategy to efficiently meet the desired quality level. The optimizer begins with an initial choice of execution strategy that uses IDJN and *SC* for each relation. As the initial strategy progresses, the optimizer derives the necessary parameters and determines a desirable execution strategy for τ_g and τ_b, while periodically (e.g., every 100 documents) checking the robustness of the new estimates using cross-validation [10], [12], [13].

A fundamental task in the optimization process is to identify the Cartesian space to explore for a given quality requirement. Exhaustively "plugging in," for each database in our output quality model in Section V, all possible values for $|Dr|$ $(0, \ldots, |D|)$ or $|Qs|$ $(0, \ldots, |Ag|+|Ab|)$ is inefficient, so instead we resort to a simple heuristic to minimize the sum of documents retrieved and processed (and hence the total execution time), conditioned on the product of the number of occurrences of good attribute values in each relation. Specifically, we aim to reduce the difference between the number of documents retrieved for each relation, since intuitively we are minimizing the sum of two numbers, conditioned on their product. Thus, we select the number of documents for each database to be as close as possible. Conceptually, this heuristic follows a "square" traversal of the Cartesian space $D_1 \times D_2$ (see Section IV).

VII. Experimental Evaluation

We now describe the experimental settings and results.

IE Systems: We trained Snowball [1] for three relations: $Executives\langle Company, CEO\rangle$, $Headquarters\langle Company, Location\rangle$, and $Mergers\langle Company, MergedWith\rangle$, to which we refer as *EX*, *HQ*, and *MG*, respectively. For θ (Section III-A), we picked *minSim*, a tuning parameter exposed by Snowball, which is the similarity threshold for extraction patterns and the terms in the context of a candidate tuple.

Data Set: We used a collection of newspaper articles from The New York Times from 1995 (NYT95) and 1996 (NYT96), and from The Wall Street Journal (WSJ). The NYT96 database contains 135,438 documents, which we used to train the extraction systems and the retrieval strategies. To evaluate the effectiveness of our approach, we used 49,527 documents from NYT96, 50,269 documents from NYT95, and 98,732 documents from WSJ. We verified that the attribute and document frequency distributions tend to be power-law for our relations.

Retrieval Strategies: For *FS*, we used a rule-based classifier created using Ripper [4]. For *AQG*, we used QXtract [2], which relies on machine learning techniques to automatically learn queries that match documents with at least one tuple. In our case, we train QXtract to only match *good* documents, avoiding the *bad* and *empty* ones.

Tuple Verification: To verify whether a tuple is good or bad, we follow the template-based approach described in [11]. Additionally, we also use a web-based "gold" set from `www.thomsonreuters.com/`.

Join Task: We defined a variety of join tasks involving combinations of the three relations and the three databases. For our discussion, we will focus on the task of computing the join *HQ* $\bowtie$ *EX*, with NYT96 and NYT95 as the hosting databases for *HQ* and *EX*, respectively.

Join Execution Strategies: To generate the join execution strategies for a task, we explore various candidates for individual relations and combine them using the three join algorithms of Section IV. For each relation, we generate single-relation strategies by using two values for *minSim* (i.e., 0.4 and 0.8) and combining each such configuration with the three document retrieval strategies.

Metrics: To compare the execution time of an execution plan chosen by our optimizer against a candidate plan, we measure the *relative difference in time* by normalizing the execution time of the candidate plan by that for the chosen plan. Specifically, we note the relative difference as $\frac{t_c}{t_o}$, where t_c is the execution time for a candidate plan and t_o is the execution time for the plan picked by the optimizer.

Accuracy of the Analytical Models: Our first goal was to verify the accuracy of our analysis in Section V. For this, we assumed perfect knowledge of the various database-specific parameters: we used the actual frequency distributions for each attribute along with the values for $|Dg|$, $|Db|$, and $|D_e|$ for each database. Given a join execution strategy, we first estimate the output quality of the join, i.e., $E[|\mathbf{Tgood}_{\bowtie}|]$ and $E[|\mathbf{Tbad}_{\bowtie}|]$, using the appropriate analysis from Section V, while varying values for the number of retrieved documents from the database, i.e., $|Dr_1|$ and $|Dr_2|$. For each $|Dr_1|$ and $|Dr_2|$ value, we measure the *actual* output quality for an execution strategy. Figure 9 shows the actual and the estimated values for the good (Figure 9(a)) and the bad (Figure 9(b)) join tuples generated using IDJN, *Scan* for both relations, and *minSim* = 0.4. Similarly, Figure 10 shows the same results for OIJN when using *Scan* for the outer relation and *minSim* = 0.4 for both relations. Then, Figure 11 compares the estimated and the actual values for ZGJN, for *minSim* = 0.4. We performed similar experiments for all other execution strategies. Additionally, we also examined the accuracy of the estimated number of documents for query-based join algorithms, i.e., for OIJN and ZGJN. Figure 12 shows the expected and the actual number of documents retrieved for varying number of queries issued

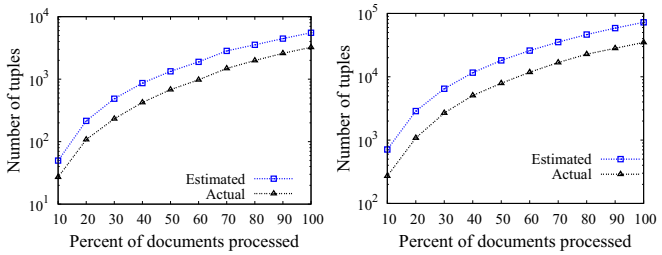


Fig. 9. Estimated and actual number of (a) good tuples and (b) bad tuples for *HQ* ⋈ *EX*, using IDJN with *Scan* and *minSim* = 0.4.

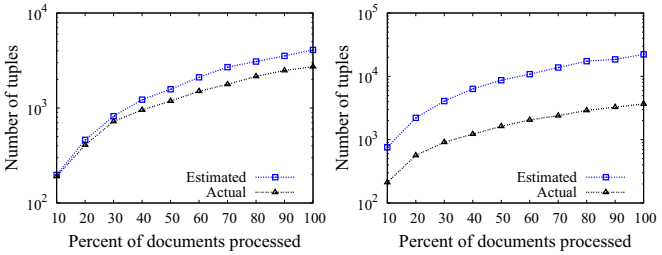


Fig. 10. Estimated and actual number of (a) good tuples and (b) bad tuples for *HQ* ⋈ *EX*, using OIJN with *Scan* and *minSim* = 0.4.

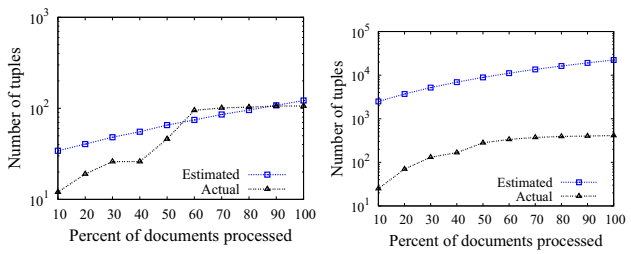


Fig. 11. Estimated and actual number of (a) good tuples and (b) bad tuples for *HQ* ⋈ *EX*, using ZGJN with *minSim* = 0.4.

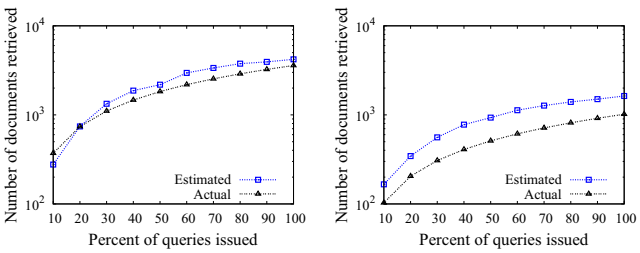


Fig. 12. Estimated and actual number of documents retrieved for (a) *HQ* and (b) *EX* for *HQ* ⋈ *EX*, using ZGJN with *minSim* = 0.4.

to each database, for ZGJN.

Overall, our estimates confirm the accuracy of our analysis. Of these observations, we discuss the case for bad tuples for OIJN (Figure 10(b)) and ZGJN (Figure 11(b)), where our model overestimates the number of bad tuples. This overestimation can be traced to a few outlier cases. To gain insight into this, we compared the expected and the actual number of bad attribute occurrences. We observed four main cases where our estimated values were more than two orders of magnitude greater than the actual values. These attribute values frequently appeared in the database but were not extracted by the extraction system at the *minSim* setting used in our experiments. For instance, one such bad attribute occurrence, "CNN Center," appears 895 and 2765 times in *HQ* and *EX*, respectively. When using OIJN and processing 50% of the database documents for the outer relation, our estimated frequencies of the bad occurrences of this value was 28.3 and 29.7 times, respectively; in reality, this attribute value was not extracted, thus resulting in an overestimate of 812 join tuples. This difference is further expanded for ZGJN due to a modeling choice: we assume that all queries used in ZGJN will match some documents and the execution will not *stall*. We can account for stalling by incorporating the reachability of a ZGJN execution based on the single-relation analysis in [10]. Overall, while our estimates have non-negligible absolute errors, they identify the actual value trends appropriately, and hence allow our query optimization approach to pick desirable execution plans for a range of output quality requirements, as discussed next.

Effectiveness of the Optimization Approach: After verifying our modeling, we studied the effectiveness of our optimization approach, which uses our models along with the parameter estimation process outlined in Section VI. Specifically, we examine whether the optimizer picks the fastest execution strategy for a given output quality requirement. For this, we provided the optimizer with the two thresholds, τ_g and τ_b. We report results for values of τ_g and τ_b for which the optimizer picked a satisfactory plan (i.e., the chosen execution matched the τ_g and τ_b quality requirements). In the future, we will systematically examine our optimizer's ability to identify scenarios where no plan can match the quality requirements (e.g., as would likely be the case for, say, $\tau_b = 0$), as well as those cases where our optimizer incorrectly predicts that no plan could match the quality requirements.

To evaluate the choice of execution strategy for a specified τ_g and τ_b pair, we compare the execution time for the chosen plan S against that of the alternate executions plans that also meet the τ_g and τ_b requirements. Table II shows the results for *HQ* ⋈ *EX*, for varying τ_g and τ_b. For each τ_g and τ_b pair, we show the number of candidate plans that meet the τ_g and τ_b requirement. Furthermore, we show the number of candidate plans that result in faster executions than the plan chosen by our optimizer and the number of candidate plans that result in slower executions than the chosen plan. Finally, to highlight the difference between the associated execution times, we show the range of relative difference in time for both faster and slower execution plans.

As shown in the table, our optimizer selects OIJN for low values of τ_g and τ_b, and progresses towards selecting IDJN coupled with *AQG* or *FS*, eventually picking IDJN coupled with *SC* for high values of τ_g and τ_b. For most cases, our optimizer selects an execution strategy that is the fastest strategy or close to the fastest strategy, as indicated by having either no candidates with faster executions than the chosen plan or a small number of such executions. For cases where the chosen plan is not the fastest option, the execution time of the faster candidates is close to the one of the chosen plan, as indicated by the relative difference values (e.g., a value of 1 indicates the execution times for both the candidate and the chosen plans were identical). An important observation is that the plans

Criteria		Candidate plans	Chosen plan					# Faster plans	# Slower plans	Relative time range for faster plans		Relative time range for slower plans	
τ_g	τ_b		JN	θ_1	θ_2	X_1	X_2			*min*	*max*	*min*	*max*
1	20	46	OIJN	0.4	0.4	FS	*(OIJN)*	5	36	0.68	0.80	1.20	27.48
2	30	46	OIJN	0.8	0.4	AQG	*(OIJN)*	10	32	0.19	0.75	1.78	11.91
2	50	47	OIJN	0.8	0.4	AQG	*(OIJN)*	11	33	0.19	0.75	1.78	11.91
4	20	39	OIJN	0.4	0.4	FS	*(OIJN)*	3	29	0.34	0.34	1.59	35.76
4	40	42	OIJN	0.4	0.4	FS	*(OIJN)*	3	37	0.34	0.34	1.59	35.76
8	40	40	OIJN	0.8	0.4	AQG	*(OIJN)*	3	33	0.19	0.19	1.15	22.20
8	80	44	OIJN	0.8	0.4	AQG	*(OIJN)*	4	38	0.19	0.19	1.15	22.20
16	50	26	IDJN	0.4	0.4	FS	AQG	-	21	-	-	1.22	11.62
16	80	36	IDJN	0.4	0.4	FS	AQG	3	30	0.66	0.94	1.10	11.62
16	160	39	IDJN	0.4	0.4	FS	AQG	3	34	0.66	0.94	1.10	11.62
32	84	26	IDJN	0.4	0.4	FS	AQG	-	22	-	-	1.26	13.30
32	160	36	OIJN	0.8	0.4	AQG	*(OIJN)*	-	35	-	-	1.55	20.62
32	320	40	OIJN	0.8	0.4	AQG	*(OIJN)*	-	39	-	-	1.55	20.62
64	320	35	IDJN	0.8	0.4	AQG	AQG	-	34	-	-	1.50	27.16
64	640	41	IDJN	0.8	0.4	AQG	AQG	-	40	-	-	1.50	27.16
128	640	21	IDJN	0.4	0.4	FS	AQG	-	20	-	-	1.19	9.41
128	1280	26	IDJN	0.4	0.4	FS	AQG	-	25	-	-	1.19	9.41
256	1280	14	IDJN	0.4	0.4	SC	AQG	-	13	-	-	1.18	2.89
256	2560	18	IDJN	0.4	0.4	SC	AQG	-	17	-	-	1.01	2.89
512	1024	1	IDJN	0.8	0.8	SC	SC	-	-	-	-	-	-
512	2560	3	IDJN	0.8	0.4	SC	SC	-	2	-	-	1.02	1.15
512	5120	4	IDJN	0.4	0.4	FS	SC	-	3	-	-	1.46	1.69
1024	5120	2	IDJN	0.8	0.4	SC	SC	1	-	0.99	0.99	-	-
1024	10240	2	IDJN	0.8	0.4	SC	SC	1	-	0.99	0.99	-	-

TABLE II

CHOICE OF EXECUTION STRATEGIES FOR DIFFERENT τ_g AND τ_b COMBINATIONS (SECTION III-C), AND COMPARING THE EXECUTION TIME OF THE CHOSEN STRATEGY AGAINST THAT OF ALTERNATIVE EXECUTION STRATEGIES THAT ALSO MEET THE τ_g AND τ_b REQUIREMENTS, FOR $HQ \bowtie EX$.

eliminated by the optimizer were an order of magnitude (10 to 35 times) slower than the chosen plans.

An intriguing outcome of our experiments is that the choices for execution strategies do not involve ZGJN. Interestingly, for our test data set, ZGJN is not a superior choice of execution algorithm as compared to other algorithms. Intuitively, ZGJN does not specifically focus on filtering out any bad documents; therefore, ZGJN does not meet the quality requirements as closely as other query-based strategies that use IDJN or OIJN along with *AQG* or *FS*. Furthermore, the maximum number of tuples that can be extracted using ZGJN is limited, which makes it a poor choice for higher values of τ_g and τ_b. ZGJN would be a competing choice for scenarios involving databases that only provide query-based access (e.g., search engines or hidden-Web databases) and also for cases where the generated queries match a relatively large number of good documents. Extending ZGJN to derive queries that focus on good documents remains interesting future work.

VIII. CONCLUSIONS

We addressed the important problem of optimizing the execution of joins of relations extracted from natural language text. As a key contribution of our paper, we developed rigorous models to analyze the output quality of a variety of join execution strategies. We also showed how to use our models to build a join optimizer that attempts to minimize the time to execute a join while reaching user-specified result quality requirements. We demonstrated the effectiveness of our optimizer for this task with an extensive experimental evaluation over real-world data sets. We also established that the analytical models presented in this paper demonstrate a promising direction towards building fundamental blocks for processing joins involving information extraction systems.

IX. ACKNOWLEDGMENTS

This material is based upon work supported by a generous gift from Microsoft Research, as well as by the National Science Foundation under Grants No. IIS-0811038, IIS-0643846, and IIS-0347903. The third author gratefully acknowledges the support of an Alfred Sloan fellowship, an IBM Faculty Award, and grants from Yahoo and Microsoft.

REFERENCES

[1] E. Agichtein and L. Gravano. Snowball: Extracting relations from large plain-text collections. In *DL*, 2000.
[2] E. Agichtein and L. Gravano. Querying text databases for efficient information extraction. In *ICDE*, 2003.
[3] W. Cohen and A. McCallum. Information extraction from the World Wide Web (tutorial). In *KDD*, 2003.
[4] W. W. Cohen. Learning trees and rules with set-valued features. In *IAAI*, 1996.
[5] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: An architecture for development of robust HLT applications. In *ACL*, 2002.
[6] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction (tutorial). In *SIGMOD*, 2003.
[7] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. In *Natural Language Engineering*, 2004.
[8] R. Gupta and S. Sarawagi. Curating probabilistic databases from information extraction models. In *VLDB*, 2006.
[9] P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.
[10] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. Towards a query optimizer for text-centric tasks. *ACM Transactions on Database Systems*, 32(4), Dec. 2007.
[11] A. Jain, A. Doan, and L. Gravano. Optimizing SQL queries over text databases. In *ICDE*, 2008.
[12] A. Jain and P. G. Ipeirotis. A quality-aware optimizer for information extraction. *ACM Transactions on Database Systems*, 2009. To appear.
[13] A. Jain, P. G. Ipeirotis, A. Doan, and L. Gravano. Join optimization of information extraction output: Quality matters! Technical Report CeDER-08-04, New York University, 2008.
[14] I. Mansuri and S. Sarawagi. A system for integrating unstructured data into relational databases. In *ICDE*, 2006.
[15] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2), Aug. 2001.
[16] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE*, 2008.
[17] W. Shen, A. Doan, J. Naughton, and R. Ramakrishnan. Declarative information extraction using Datalog with embedded extraction predicates. In *VLDB*, 2007.