**Literature survey:**

The World-Wide Web consists of a huge number of unstructured documents, but it also contains structured data in the form of HTML tables. We extracted 14.1 billion HTML tables from Google’s general-purpose web crawl, and used statistical classification techniques to find the estimated 154M that contain high-quality relational data. Because each relational table has its own “schema” of labeled and typed columns, each such table can be considered a small structured database. The resulting corpus of databases is larger than any other corpus we are aware of, by at least five orders of magnitude. We describe the WebTables system to explore two fundamental questions about this collection of databases. First, what are effective techniques for searching for structured data at search-engine scales? Second, what additional power can be derived by analyzing such a huge corpus? First, we develop new techniques for keyword search over a corpus of tables, and show that they can achieve substantially higher relevance than solutions based on a traditional search engine. Second, we introduce a new object derived from the database corpus: the attribute correlation statistics database (AcsDB) that records corpus-wide statistics on cooccurrences of schema elements. In addition to improving search relevance, the AcsDB makes possible several novel applications: schema auto-complete, which helps a database designer to choose schema elements; attribute synonym finding, which automatically computes attribute synonym pairs for schema matching; and join-graph traversal, which allows a user to navigate between extracted schemas using automatically-generated join links. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the H.3 [Information Storage and Retrieval]: Online Information Services; H.2 [Database Management]: Miscellaneous The Web is traditionally modelled as a corpus of unstructured documents. Some structure is imposed by hierarchical URL names and the hyperlink graph, but the basic unit for reading or processing is the unstructured document itself. However, Web documents often contain large amounts of relational data. For example, the Web page shown in Figure 1 contains a table that lists American presidents1 . The table has four columns, each with a domain-specific label and type (e.g., President is a person name, Term as President is a date range, etc) and there is a tuple of data for each row. This Web page essentially contains a small relational database, even if it lacks the explicit metadata traditionally associated with a database. We extracted approximately 14.1 billion raw HTML tables from the English documents in Google’s main index, and used a series of techniques to recover those tables that are high-quality relations [6]. Recovery involves filtering out tables that are used for page layout or other non-relational reasons, and detecting labels for attribute columns. We estimate that the tested portion of our general web crawl contains 154M distinct relational databases - a huge number, even though it is just slightly more than 1.1% of raw HTML tables. Previous work on HTML tables focused on the problem of recognizing good tables or extracting additional information from individual tables [27, 30, 32]. In this paper we consider a corpus of tables that is five orders of magnitude larger than the largest one considered to date [27], and address two fundamental questions: (1) what are effective methods for searching within such a collection of tables, and (2) is there additional power that can be derived by analyzing such a huge corpus? We describe the WebTables system that explores these questions. The main motivation for searching such a corpus of tables is to enable analysis and integration of data on the Web. In particular, there is a recent flurry of tools for visualizing structured data and creating mashups on the Web (e.g., many-eyes.com swivel.com, Yahoo Pipes, Microsoft Popfly). Users of such tools often search the Web for good tabular data in a variety of domains. In addition, while searches for structured data may account for only a small fraction of general web searches, a scan over a 1-day log of Google’s queries revealed that for close to 30 million queries, users clicked on results that contained tables from our filtered relational corpus, which is quite substantial. Document search engines are commonplace, and researchers have studied the problem of keyword ranking for individual tuples within a database [2, 15]. However, to perform relation ranking, i.e., to sort relations by relevance in response to a user’s keyword search query, WebTables must solve the new problem of ranking millions of individual databases, each with a separate schema and set of tuples. Relation ranking poses a number of difficulties beyond web document ranking: relations contain a mixture of “structural” and related “content” elements with no analogue in unstructured text; relations lack the incoming hyperlink anchor text that helps traditional search; and PageRank-style metrics for page quality are useless when tables of widely-varying quality can be found on the same web page. Finally, relations contain text in two dimensions and so many cannot be efficiently queried using the standard inverted index. We describe a ranking method that combines table-structureaware features (made possible by the index) with a novel query-independent table coherency score that makes use of corpus-wide schema statistics. We show that this approach gives an 85-98% improvement in search quality over a na¨ıve approach based on traditional search engines. To validate the power of WebTables’s corpus, we describe the attribute correlation statistics database, (ACSDb), which is a set of statistics about schemas in the corpus. In addition to improving WebTables’s ranking, we show that we can leverage the ACSDb to offer unique solutions to schema-level tasks. First, we describe an algorithm that uses the ACSDb to provide a schema auto-complete tool to help database designers choose a schema. For example, if the designer inputs the attribute stock-symbol, the schema auto-complete tool will suggest company, rank, and sales as additional attributes. Unlike set-completion (e.g., Google Sets) that has been investigated in the past, schema autocomplete looks for attributes that tend to appear in the same schema (i.e., horizontal completion). Second, we use the ACSDb to develop an attribute synonym finding tool that automatically computes pairs of schema attributes that appear to be used synonymously. Synonym finding has been considered in the past for text documents [16], but finding synonyms among database attributes comprises a number of novel problems. First, databases use many attribute labels that are nonexistent or exceedingly rare in natural language, such as abbreviations (e.g., hr for home run) or non-alphabetic sequences (e.g., tel-#); we cannot expect to find these attributes in either thesauri or natural text. Second, the context in which an attribute appears strongly affects its meaning; for example, name and filename are synonymous, but only when name is in the presence of other file-related attributes. If name is used in the setting of an address book, it means something quite different. Indeed, two instances of name will only be synonymous if their co-attributes come from the same domain. We give an algorithm that automatically detects synonymy with extremely high accuracy. For example, our synonym-finder takes an input domain and gives an average of four correct synonym pairs in its first five emitted pairs. Finally, we show how to use the ACSDb for join-graph traversal. This tool can be used to build a “schema explorer” of the massive WebTables corpus that would again be useful for database designers. The user should be able to navigate from schema to schema using relational-style join links (as opposed to standard hypertext links that connected related documents). Our extracted tables lack explicit join information, but we can create an approximation by connecting all schemas that share a common attribute label. Unfortunately, the resulting graph is hugely “busy”; a single schema with just two or three attributes can link to thousands of other schemas. Thus, our set of schemas is either completely disconnected (in its original state) or overly-connected (if we synthesize links between attribute-sharing schemas). It would be more useful to have a graph with a modest number of meaningful links. To address this problem, we introduce an ACSDbbased method that clusters together related schema neighbors. All of the above tools are examples of how web-scale data can be used to solve problems that are otherwise very hard. They are similar in spirit to recent efforts on machine translation [4] and spell-correction that leverage huge amounts of data. The distinguishing feature of the ACSDb is that it is the first time such large statistics have been collected for structured data schema design. We note that the idea of leveraging a large number of schemas was initially proposed in [17] for the improving schema matching. Our work is distinguished in that we consider a corpus that is several orders of magnitude larger, and we leverage the corpus more broadly. Our synonym finder can be used for schema matching, but we do not explore that here. Before we proceed, we distinguish between the data we manage with WebTables and the deep web. The WebTables system considers HTML tables that are already surfaced and crawlable. The deep web refers to content that is made available through filling HTML forms. The two sets of data overlap, but neither contains the other. There are many HTML tables that are not behind forms (only about 40% of the URLs in our corpus are parameterized), and while some deep-web data is crawlable, the vast majority of it is not (or at least requires special techniques, such as those described in [14]). In contrast to the work we describe in this paper, deep web research questions focus on identifying high quality forms and automatically figuring out how to query them in a semantically meaningful fashion. In addition to HTML tables and the deep web, there are many kinds of structure on the Web, including tagged items, ontologies, XML documents, spreadsheets, and even extracted language parses [19]. In this paper we will only consider the table tag. This paper focuses on the extracted table corpus, how to provide search-engine-style access to this huge volume of structured data, and on the ACSDb and its applications. We do not study how to match or integrate the table data, though we have done so elsewhere [6]. The remainder of this paper is organized as follows. Section 2 describe our basic model and the ACSDb. In Section 3, we describe how to rank tables in response to keyword query on WebTables. Section 4 covers our three novel ACSDb applications: schema auto-complete, attribute synonym finding, and join-graph discovery. We present experimental evaluations in Section 5, and conclude with discussions of related and future work (Sections 6 and 7). We begin by describing the set of relations we extract from the web crawl, and the statistics we store in the attribute correlation statistics database (ACSDb). The database corpus that is contained within the raw HTML tables is hugely valuable, containing data drawn from millions of sites and across a vast array of topics. We wrote a system, described in [6], that combines hand-written detectors and statistically-trained classifiers to filter good relations from bad ones. There is no strict method for distinguishing a “truly relational” HTML table from a nonrelational one; we tested our detector’s output on a humanmarked test sample to measure how well it performs. After filtering out non-relational HTML tables, we used a separate trained detector to extract any embedded metadata (such as the first row in the table in Figure 1). The extraction pipeline is pictured in Figure 2. We tuned the relation-filter for relatively high recall and low precision, so that we lose relatively few true relations. We tuned the metadata-detector to equally weigh recall and precision. Table 1 summarizes the performance of the extractor, which yields more than 125M high-quality relations from our original web crawl. The precision and recall of the extractor is roughly similar to other domain-independent information extraction systems (only Relational Filtering precision is relatively low, but we specifically maximized recall at the cost of this precision). involved, and is not germane to our current discussion. Here we simply assume we have a corpus, R, of databases, where each database is a single relation. For each relation, R ∈ R, we have the following: In 2008, we wrote about WebTables, an effort to exploit the large and diverse set of structured databases casually published online in the form of HTML tables. The past decade has seen a flurry of research and commercial activities around the WebTables project itself, as well as the broad topic of informal online structured data. In this paper, we1 will review the WebTables project, and try to place it in the broader context of the decade of work that followed. We will also show how the progress over the past ten years sets up an exciting agenda for the future, and will draw upon many corners of the data management community. In 2008, the Web had been traditionally modelled as a corpus of unstructured documents. Some structure was imposed by hierarchical URL names and the hyperlink graph, but the basic unit for reading or processing was the unstructured document itself. That is mostly still true in 2018. However, Web documents often contain large amounts of relational data. For example, the Web page shown in Figure 1 (from the original paper [7]) contains a table that lists 1 One of the things that has made the WebTables project exciting is its long lifespan: the project began in 2007, and is still ongoing. The authors of this paper took an especially large role in WebTables, but not everyone who worked on the project is an author of this paper or even necessarily known to us. We are very grateful to everyone who has contributed over the years. American presidents. The table has four columns, each with a domain-specific label and type (e.g., President is a person name, Term as President is a date range, etc) and there is a tuple of data for each row. This Web page essentially contains a small relational database, even if it lacks the explicit metadata traditionally associated with a database. The goal of the original WebTables paper [7] was to automatically detect these “database-like” HTML tables, use them to construct the largest corpus of databases to date, and then build novel applications out of the resulting corpus. We extracted 14.1B HTML tables from Google’s generalpurpose web crawl, and then used a trained classifier to identify the estimated 154M tables that were database-like. The percentage of raw tables that described databases was small — about 1.1% — but the number of raw tables was so large that the resulting corpus of databases was still larger than any previously-known collection, by at least five orders of magnitude. The paper described how we constructed the corpus, and described a number of novel applications that the corpus enabled. The WebTables work at Google was very exciting because it combined the Web’s enormous scale and topical reach with questions that have been traditionally associated with relational databases, such as how to understand and recover schema information. However, we did not imagine the many positive events that would follow. Many subsequent research papers, from both academia and industry, improved the original extraction methods, then applied the resulting datasets to novel problems such as attribute discovery and entity extension. There was also industrial software engineering effort – most notably at Google and Microsoft – that built real products similar to the use cases described in the original WebTables paper, as well as entirely novel ones. However, we also believe there are still tremendous opportunities around extracting and manipulating structured data on the Web. Indeed, we think the next decade holds even more promise for WebTables-style work than the last. In this paper, we will offer a brief summary of the original WebTables work, and attempt to describe and organize much of the intellectual work that followed. We will also describe the practical engineering efforts that were necessary to turn the WebTables vision into real products. Finally, we will sketch a vision of what we believe are open opportunities around WebTables-like work, both in the intellectual sphere and the practical engineering one. WebTables paper [7] and its core contributions. 1.1.1 Extraction and Data Model We collected roughly 14.1 billion HTML tables from the Google search web crawl, and applied a trained is-relational classifier to identify the tables as relational or non-relational. The relational label is somewhat informal, defined driven by human judgmements: the human judges wanted tables where rows clearly represent separate tuple-like objects, and columns represent different dimensions of each tuple. Extraction details were described in Cafarella, et al. [8]. A “header row” of attribute labels at the top of the table is optional, but if recovered offers a small amount of schema-like information about the extracted data table. We trained a second classifier to detect this header row. Of course the schema information is extremely informal, and even if recovered is not very expressive. For example, even attribute typing information is not explicit, and many traditional relational schema elements such as key constraints are missing. We applied the is-relational classifier to the collection of raw HTML tables and obtained a corpus R of databases (where the classifier returned a verdict of relational). Each database consists of a single relation. For each relation R ∈ R, we have: • The url and page offset where R was recovered; these uniquely identify R. • The header row, or “schema” RS , which is an ordered list of attribute labels. For example, the table in Figure 1 has the attributes RS = [President, Party, ...]. One or more elements of RS may be empty strings (e.g., if the table’s schema cannot be recovered). • A list of tuples, RT . A tuple t is a list of data strings. The size of a tuple t is always |RS |, though one or more elements of t may be empty strings. Extracting a large collection of database-like tables and their relational attribute labels allowed us to build the attribute correlation statistics database, or ACSDb. It contains statistics about general WebTables schema use. The ACSDb listed each unique schema S found in the set of all RS , along with a count that indicates how many relations contain the given S. We assume two schemas are identical if they have the same set of attributes (regardless of order). The ACSDb A is a set of pairs of the form (S, c), where S is a schema of a relation in R, and c is the number of relations in R that have the schema S. We only count one schema per internet domain name, to prevent a single site with many similar pages from swamping the counts. The resulting ACSDb contained 5.4M unique attribute labels, and 2.6M unique schemas. Unsuprisingly, a relatively small number of schemas appear very frequently, while most schemas are rare. The ACSDb was simple, but critically allowed us to compute the probability of seeing various attributes in a schema. For example, p(address) is simply the sum of all counts c for pairs whose schema contains address, divided by the total sum of all counts. We can also detect relationships between attribute names by conditioning an attribute’s probability on the presence of a second attribute. For example, we could compute p(address|name) by counting all the schemas in which “address” appears along with “name”, and normalizing by the counts for seeing “name” alone. We used this corpus of databases to build several applications. The first was a simple keyword search tool: it would take a user’s search query as input, and return a ranked list of WebTables-derived databases to answer the query. Our initial paper used “city population” as an example query. Subsequent work at Google deployed this idea in the core search engine; Figure 2 shows that query from the original paper’s screenshot, and that same query on the Google search engine of today. We also built several applications on top of the ACSDb. Two of the most interesting are schema autocomplete and attribute synonym finding. The schema autocomplete tool helped database designers build a schema, by suggesting the most-likely next attributes to add to a schema. For example, if the designer inputs the attribute stock-symbol, the ACSDb-powered schema auto-complete tool will suggest company, rank, and sales as additional attributes. This tool worked by using ACSDb statistics to find the most probable attributes, conditioned on seeing attributes the designer has already entered. The attribute synonym finding tool automatically computed pairs of schema attributes that seem to be used synonymously in the WebTables corpus. For example, the tool could find that hr and home run are used synonymously when representing baseball data, even though this would be an extremely unlikely pair of words to find in a traditional thesaurus or other linguistic resource. This tool worked by identifying pairs of attribute labels where (1) the labels never appeared together in the same WebTables schema, and (2) the labels frequently share similar co-attributes, among other sources of evidence. In 2008, WebTables was not the first effort to identify databases from HTML tables. Other systems [10, 37, 41], especially that of Gatterbauer, et al. [18], had tried a number of interesting approaches for extracting databases from HTML pages. WebTables was the first system we know of to obtain a database corpus at its scale. Keyword search over structured data was a known problem in the database literature, but tended to focus on returning tuples from a single large database, as with the DBXplorer [2] and DISCOVER [22] projects. Keyword-driven web search engines sometimes returned structured results, but were usually limited to a tiny number of domains, such as weather. In 2008, most other access paths for popular consumption of structured data, such as mobile device app stores and voice assistants, did not yet exist The WebTables project was one example of research into online structured data – a long-lasting line of intellectual and engineering work that includes Semantic Web [4], Web of Linked Data [5], and many other projects. However, dedicated research into Web-embedded data tables per se has became quite popular in the last decade. One early direction was to extend table extraction beyond HTML tags. The original paper focused on identifying very conventional tables of data, with attribute-oriented columns and tuple-oriented rows. However, relational data are also encoded as attribute-value pairs in the form of vertical tables (e.g., Wikipedia infoboxes), or as entries in lists [16, 13]. For example, a list of cartoons may have entries such as “Duck Amuck (Warner Bros./1953)”. This string encodes a structured record containing the title, the producer, and the release year. Elmeleegy et al. [16] introduced a pipeline that splits individual list entries into candidate rows, performs attribute column alignment across rows, and finally refines the table to address inconsistencies. Chu et al. [13] further introduced syntactic and semantic coherence measures for list extraction, and learned semantic coherence measures by leveraging column co-occurence statistics from existing web tables. Several researchers produced web tables from the public Common Crawl [1, 24, 15], thereby making them available to a broad audience outside the large Web companies. Wang, et al. [36] improved extraction quality by leveraging curated knowledge bases Table Search was a core and exciting potential application in the WebTables paper, and it is not surprising that it attracted substantial research attention. In general, keyword-based table search ranks extracted tables based on a combination of cell contents, attribute names, and surrounding text (e.g., page title). However, this approach may not be sufficient for several important classes of search queries, encountered by Chakrabarti, et al. when deploying a web table service on the Bing search engine [9]. One is row-subset queries, which only request a subset of rows in a larger table. For example, the query “largest software companies in USA” may only match a table of large software companies, and must be filtered to the subset of USA rows. Another is entity-attribute queries, such as “aberdeen population”, which match some keywords to entities (“aberdeen”) and other keywords to attribute names (“population”). Another way to navigate a set of tables is via column search. This method does not use keywords, except perhaps in the initial user interaction. Instead, users begin with an ”initial table,” then navigate through a “concept space” of tables that would be good join candidates for the original. Different methods built this space using knowledge bases [17], isA databases [32], or crowd sourcing [17]. In several “table enhancement” projects, the user submits a query table to the system, which then attempts to “complete” it by filling in attribute values, recommending novel attributes, or adding more entities of the same type. Several operators have been proposed for specific types of table enhancement tasks [6, 39, 43]. In Augmentation by attribute name (ABA) (also called EXTEND() in [6]), the system fills in attribute values given an explicit user-provided attribute. In augmentation by example (ABE), the system fills in attribute values given other examples of desired attribute values (but no attribute name). The Attribute Discovery (AD) operator recommends important attributes given a list of entity names. Cafarella, et al. [6] initially implemented EXTEND() with a combination of schema matching and search engine results to find candidate completion values from extracted data tables. InfoGather [39] observed that precision and coverage of the filled-in values can be improved by exploiting indirect match tables that are a “hop away” from the original query table. This method was used to implement ABA, ABE, and AD. The follow-on InfoGather+ [43] system ensured that the web tables used for EXTEND() contained attributes were semantically consistent. For instance, extending a table of companies with “revenue” should not draw from columns with names such as “2010 revenue”, “2011 revenue”, and “revenue (euros)” even though they are textually quite similar to the original attribute name. Another example of table enhancement is fact lookup, in which the query table contains a single entity and single attribute, and the goal is to retrieve the attribute value (the fact). To answer such queries, FACTO [40] identified web pages about individual entities (e.g., a wikipedia page about Barack Obama), then found attribute-value tables from the relevant pages, and finally retrieved answers from them. Concept expansion takes as input a high level concept (e.g., rock stars) and an example list of entities within the concept (e.g., freddy mercury, yoshiki, and prince) and expands the set of entities within the concept. Wang, et al. [34] identified possible source-data web tables for this task by examining the surrounding text (e.g., table captions), and employing only exclusive tables to avoid semantic drift. The user-provided examples serve to seed this iterative process. Chen, et al. attempted to solve a similar task, with a system tailored for long-tail infrequently-observed items [11]. One important aspect of table improvement is to improve the quality of table-contained data. Along this line, Wang et. al. [35] proposed a framework based on functional dependencies (FDs). Unlike in traditional database design, where FDs are specified as statements of truth about all possible instances of the database; in web environment, FDs are not specified over the data tables. Instead, FDs are extended with probabilities to capture their inherent uncertainty, and are generated using counting-based algorithms over many data sources. These probabilistic FDs can improve data and schema quality by (1) pinpointing dirty data sources and (2) normalizing large mediated schemas. Many of the above techniques fundamentally leverage relationships among columns, rows, and tables, often with a machine learning component. Limaye, et al. [25] studied how to annotate web tables with the presence of entities in cells, attribute types and concepts; and the presence of relationships between attributes. They used a graphical model that jointly learns these annotations within a single model, and leverages the YAGO [31] knowledge base as a source of attribute concepts and entities. Pimplikar, et al. [27] employed a similar joint-labeling technique by modeling table enhancement as a graphical model. The relationship between tables can be further leveraged to synthesize tables that are the results of combining multiple related tables and thereby generating data tables that are not present anywhere on the Web. Ling, et al. [26] explored how tables with the same semantics can be discovered and vertically combined into a table with a more complete set of rows. Das Sarma, et al. [30] looked at tables that share core identifying attributes and yet complement each other on other attributes; these therefore can be horizontally combined to provide a table with a more complete set of columns. Record linkage clusters records such that each cluster corresponds to a single distinct real-world entity. It is a crucial step in data cleaning and data integration. In the big data era, the velocity of data updates is often high, quickly making previous linkage results obsolete. This paper presents an end-to-end framework that can incrementally and efficiently update linkage results when data updates arrive. Our algorithms not only allow merging records in the updates with existing clusters, but also allow leveraging new evidence from the updates to fix previous linkage errors. Experimental results on three real and synthetic data sets show that our algorithms can significantly reduce linkage time without sacrificing linkage quality. Record linkage (surveyed in [8]) clusters database records such that each cluster corresponds to a single distinct real-world entity (e.g., a business, a person). It is a crucial step in data cleaning and data integration. The big data era raises two challenges for record linkage. First, the volume of data is often huge and applying record linkage usually takes a long time. Second, the velocity of data updates is often high, quickly making previous linkage results obsolete. These challenges call for an incremental strategy, such that we can quickly update linkage results when data updates arrive. There are two goals for incremental linkage. First, we wish that the incremental approach obtains the same or very similar results as applying batch linkage. Second, we wish to conduct incremental linkage significantly faster than batch linkage. A natural thought for incremental linkage is that for each inserted record, we compare it with existing clusters, then either put it into an existing cluster (i.e., referring to an already known entity), or create a new cluster for it (i.e., referring to a new entity). However, every linkage algorithm may make mistakes and the extra information from the data updates can often help us identify and fix such mistakes, as we illustrate next with an example. EXAMPLE 1.1. Figure 1(a) shows a set of 10 business records that represent 5 businesses. For the purpose of illustration, we compute pairwise similarity in a simple way: we compare (1) name, (2) street address excluding house number, (3) house number in street address, (4) city, and (5) phone; the similarity is 1 if all five values are the same, .9 if four are the same, .8 if three are the same, and 0 otherwise. Figure 1(b) shows the similarity graph between the records, where each node represents a record and each edge represents the pairwise similarity. It also shows the results of correlation clustering (we describe it in Section 2) as the linkage result. Note that it wrongly clusters r4 with r1 − r3 because of the wrong phone number from r4 (in italics); it fails to merge r5 and r6 because of the missing information in r6; and it wrongly merges r9 with r7 − r8 instead of with r10, because r9 appears similar to r7 − r8 while r10 does not (different name, different house number, and missing phone). Now consider four updates ∆D1 − ∆D4 (Figure 2(a)); they together insert records r11 − r17. Figure 3 shows the updated similarity graph and the results of the aforementioned naive approach. It creates a new cluster for r11 as it is different from any existing record, and adds the rest of the inserted records to existing clusters. However, a more careful analysis of the inserted nodes allows fixing some previous mistakes and obtaining a better clustering (shown in Figure 2(b)). First, because r12 − r13 are similar both to r5 and to r6, they provide extra evidence to merge r5 and r6. Second, when we consider r1 − r4, r14 − r15 jointly, we find that r1 − r3, r14 − r15 are very similar, but r4 is different from most of them, suggesting moving r4 out. Third, with r16 − r17, r9 appears to be more similar to r10 and r16 than to r7 − r8, suggesting moving r9 from C4 to C5. Incremental record linkage has been studied before in [12, 13], where the main focus is the case when the matching rules evolve over time. In [13] the authors briefly discussed the case of evolving data and identified a general incremental condition under which incremental linkage can be easily carried out using the batch linkage method. This condition requires that for any arbitrary subset of records and its batch clustering results, if we treat each of the rest of the records as a singleton cluster and apply the same algorithm on all of the resulting clusters, we obtain exactly the same results as we apply the algorithm directly on all singleton clusters. As an example, agglomerative clustering, which iteratively merges similar clusters, is general incremental. However, not many clustering algorithms satisfy this condition. For example, the aforementioned naive approach that iteratively adds each record into an existing clustering is order-dependent, so does not satisfy this condition. Moreover, many clustering algorithms, such as correlation clustering (we shall explain it soon), operate on records rather than subsets of records, so the batch algorithm cannot directly apply on previous clustering results. In this paper we ask two questions. First, in case the batch linkage algorithm is not general incremental, can we do better than just conducting linkage from scratch? Second, how can we make a trade-off between quality of the linkage results and efficiency of the algorithm? This paper presents a set of algorithms that can incrementally conduct record linkage when new records are inserted and when existing records are deleted or changed (i.e., values are modified). In particular, we make the following three contributions. • We describe an end-to-end solution for incremental record linkage. Our solution incrementally maintains a similarity graph for the records, and conducts incremental graph clustering, resulting in clusters of records that refer to the same real-world entity. • For incremental graph clustering, we first propose two optimal algorithms that apply clustering on subsets of the records rather than all records. We then design a greedy approach that conducts linkage incrementally in polynomial time by merging and splitting clusters connected to the updated records, and moving records between those clusters. • We instantiate our algorithms on two clustering methods that do not require knowing the number of clusters a priori and are used often in record linkage: correlation clustering and DB-index clustering. Our experiments on real-world data sets show that our algorithms run significantly faster than batch linkage while obtaining similar results. While we evaluate our approaches with tabular datasets, they apply to any entity resolution setting where entities can be modeled as nodes and similarities between entities as edges in a graph. The rest of the paper is organized as follows. Section 2 formally defines the problem and describes an end-to-end solution for incremental record linkage. Sections 3-4 describe our incremental linkage algorithms. Section 5 presents our experimental results, Section 6 discusses related work, and Section 7 concludes. This section formally defines the problem of incremental record linkage (Section 2.1). We then describe the framework for incremental linkage (Section 2.2). and review techniques for graph clustering, which is a key component in record linkage (Section 2.3). Given a set of records, record linkage is essentially a clustering problem, where each cluster contains records that correspond to a single distinct real-world entity. We denote by D a set of records and by LD a clustering of records in D as record-linkage results. Ideally, the clustering should have both high precision (i.e., records in the same cluster refer to the same real-world entity) and high recall (i.e., records referring to the same real-world entity belong to the same cluster). We denote by F the batch linkage method that obtains LD on D; that is, F(D) = LD. We consider three types of update operations: Insert adds a new record; Delete removes an existing record; and Change modifies one or a few values of an existing record. Note that Change can be achieved by first removing the old record and then inserting the new record; however, as we show later, considering Change directly can be more efficient. We call those update operations (Insert, Delete, and Change) made at the same time an increment, denoted by ∆D. We denote the result of applying ∆D to D by D + ∆D. Note that because ∆D can contain deletes and changes, the number of the resulting records may be lower than the sum of the number of original records and the number of records in the increment; that is, |D + ∆D| ≤ |D| + |∆D|. In this paper, we assume every increment ∆D is valid: the record in a Delete or Change operation already exists in D, and the record in an Insert does not exist in D. We now define incremental linkage. A basic step in integration is the identification of linkage points, i.e., finding attributes that are shared (or related) between data sources, and that can be used to match records or entities across sources. This is usually performed using a match operator, that associates attributes of one database to another. However, the massive growth in the amount and variety of unstructured and semistructured data on the Web has created new challenges for this task. Such data sources often do not have a fixed pre-defined schema and contain large numbers of diverse attributes. Furthermore, the end goal is not schema alignment as these schemas may be too heterogeneous (and dynamic) to meaningfully align. Rather, the goal is to align any overlapping data shared by these sources. We will show that even attributes with different meanings (that would not qualify as schema matches) can sometimes be useful in aligning data. The solution we propose in this paper replaces the basic schemamatching step with a more complex instance-based schema analysis and linkage discovery. We present a framework consisting of a library of efficient lexical analyzers and similarity functions, and a set of search algorithms for effective and efficient identification of linkage points over Web data. We experimentally evaluate the effectiveness of our proposed algorithms in real-world integration scenarios in several domains. Many increasingly important data management and mining tasks require integration and reconciliation (or fusion) of data that reside in large and heterogeneous data sources. Data integration is Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy. generally defined as combining data to provide users with a unified view of the data [15] whereas in data fusion, duplicates are merged and conflicting attributes values are identified and possibly repaired in order to provide a single consistent value for each data attribute [3]. Data fusion therefore involves duplicate detection, also known as Entity Resolution or record linkage, where the goal is to identify data records that refer to the same entity. The first step in a data integration or fusion system is identification of linkage points between the data sources, i.e., finding correspondences between attributes in the data sources. Traditionally, this is performed by schema matching, where the goal is to identify the schema elements of the various data sources that are semantically related. However, the massive growth in the amount of unstructured and semi-structured data in data warehouses and on the Web has created new challenges for this task. In what follows, we first describe an example real-world integration scenario and then describe the unique challenges not addressed in previous work. Consider a scenario where data about public companies is gathered from different sources on the Web. We have collected three data sets from online sources: Freebase [28], DBpedia [23], and the U.S. Securities and Exchange Commission (SEC) [29]. The data sets are respectively extracted using Freebase’s Metaweb Query Language, DBpedia’s SPARQL endpoint, and IBM SystemT [4] applied on the online SEC forms. Each of the three data sets is converted into a collection of JSON encoded records. Each JSON record is a tree representing various facts about a public company (an entity). Table 1 shows some statistics over these data sets. One can see that the three data sets are very different in structure. JSON trees in DBpedia have over 1700 different paths (or attributes), and describe 1.9 million facts (or entity-attribute-value triples), while JSON trees in SEC have only 72 distinct paths, but describe 4.5 million facts. This shows that DBpedia contains very heterogeneous records, while records in the SEC data set have a more consistent structure. Another observation is that Freebase has over 74,000 JSON records describing 1.9 million facts, while SEC has only approximately 2,000 records, but describes 4.5 million facts. The SEC data contains much more elaborate records than Freebase. The three data sets should have significant overlap due to the common topic (public companies). Despite the fact that the SEC data has the greatest structural regularity, this data does not subsume the other data sets, so alignment and use of these data sets together can provide more information than any single source. Toward data integration, we first need to identify linkage points, i.e., paths (attributes) in the JSON trees that are shared or related among the data sets and that are useful in identifying entities that can be linked (a formal definition is given in the next section). One possible approach is to apply schema matching algorithms [18] based on the schema information of the data sets. A purely schema-based matching algorithm would fail in many cases. For instance, DBpedia contains the labels dbpedia:stockSymbol, dbpedia:stockTicker and dbpedia:tickerSymbol for stock symbols.1 Further investigation of the instances reveals that each of these three attributes in DBpedia actually contain only a single value, perhaps because the DBpedia extraction algorithm has been unable to extract the stock symbols from Wikipedia. So, the label name in this case does not reflect the data, and is not useful for matching. Moreover, matching Freebase and SEC based on the ticker symbol and stockSymbol attributes results in ambiguous links (one company matched with more than one company on the other side). This happens because some (subsidiary) companies in Freebase share stock symbols with their parent companies. This shows that these stock symbol attributes are not as strong linkage points as one would expect. Even if the schema labels are meant to be representative and can be used for matching schema elements, there could be differences in data representation and style that make matching the records difficult. In fact, our experience in this and other similar scenarios (as described in Section 5) show that for the most interesting and useful linkage points, schema labels and values do not match using simple string comparison. For example, there are different attribute labels used for URLs (e.g., url in Freebase, foaf:page and foaf:homepage in DBpedia) and there are different ways of writing URLs (e.g., http://ibm.com vs. http://www.ibm.com/). Another example is different representations of identifiers, e.g., the unique identifiers in SEC, called CIKs, are fixed-length numbers such as #0000012345 stored in an attribute labeled cik but Freebase represents them as /business/cik/12345 stored in the identifier attribute id. There are also cases where only a part of the values can be used to link the records. For example, URI http://dbpedia.org/resource/Citigroup in DBpedia matches with ID /en/topic/citigroup in Freebase, and URL http://www.audi.com/ matches with name Audi. Since Citigroup and Audi are relatively rare names, (URI,ID) and (URL,name) attribute pairs can effectively be used to link these records. However, such linkage points can easily be missed unless the user has a thorough knowledge of both data sets. Gaining such knowledge could be challenging as it may require examining a large and representative portion of the data to understand when an attribute could be useful in linking a portion of the data In traditional data integration systems, identification of linkage points is performed either manually (possibly using a userinterface designed for matching schema elements) or by an automatic or semi-automatic schema matching algorithm. However, the size and heterogeneity of the schema, along with schema errors present in many sources that use automated information extraction, make many existing schema-based approaches inaccurate in aligning may data sets. In addition, the size and heterogeneity of Web data makes existing instance-based approaches [14, 22] ineffective and inefficient in aligning data. In this paper, we present a framework for identification of linkage points for multi-source Web data integration. Our framework includes a novel class of search algorithms to identify strong linkage points (that is, attributes that can be used to link entities across data sets) even when such attributes are weak schema matches. Importantly, we are specifically looking for attributes that help in identification of entities that can be linked. So unlike in schema matching, we are not interested in finding all corresponding attributes (for example, matching color and colour). As a result, our search can be much more focused. Our algorithms take advantage of 1) a library of lexical analyzers, 2) fast record-level and tokenlevel inverted indices, 3) a library of similarity functions, and 4) a set of filtering strategies to filter false-positive results of the search. We have implemented and experimentally evaluated the framework in several real world Web data integration scenarios such as the one described above. We show the effectiveness of different components of the framework in discovering linkage points in these scenarios, and how the discovered linkage points can enhance the record linkage process. Next, we present our problem definition. Section 3 presents our proposed framework, and Section 4 presents the details of the search algorithms for attribute selection and identification of linkage points. We present a thorough experimental evaluation of the search algorithms in Section 5. Sections 6, 7, and 8 conclude the paper with a summary of the results, brief overview of the related work, and a few interesting directions for future work. A lexical analyzer (or tokenizer) l is defined as a function that takes an atomic value v and converts it into a set of tokens v. Some analyzers only split the string value into a set of tokens such as word tokens or q-grams (substrings of length q of the string). Other analyzers perform string transformations (or normalization) by for example removing specific characters or changing letter case in addition to tokenization (or without any tokenization). We refer to the set of tokens of all the instance values of attribute p in record r as Instancesl r(p). Similarly, Instancesl D(p) represents the set of all the tokens of all the instance values of attribute p in data set D. A record matching function is defined as a Boolean function f(rs, rt) that returns true if the two records rs and rt match, according to a matching criteria. The matching criteria can be defined using an attribute matching function f(ps,pt)(rs, rt), that returns true if the instance values of the two attributes Instancesrs (ps) and Instancesrt (pt) are relevant. Using a Boolean relevance function r(Vs, Vt), we say that two sets of values Vs and Vt are relevant if r(Vs, Vt) is true. There are several ways to define the notion of relevance. For example, two sets Vs and Vt can be considered relevant if there exists vs ∈ Vs and vt ∈ Vt such that vs and vt are similar. Two atomic values vs and vt are considered similar if their similarity score according to a value similarity function sim() is above a threshold θ, i.e., sim(vs, vt) ≥ θ. DEFINITION 3 (LINKAGE POINT). We define a linkage point between two data sets Ds and Dt as a pair of attributes (ps, pt) such that for some attribute matching function f, the following set is non-empty: To align data, we need to find attributes containing values that when linked can help to identify related records (entities). To do this, we use two measures, strength and coverage. Given a potential linkage point (ps, pt), the strength measures how identifying the links between values of these attributes are. Strength is defined as the percentage of distinct records that can be linked by a pair of attributes. For example, in Figure 1, attribute freebase company -> founded and sec company -> director -> number of shares can be used to link the two records. However, if many other companies are founded in the same year or have the same number of shares, this linkage set will also link these records with several other records and therefore form a linkage point with low strength. The coverage measures how many records are linkable. While high coverage is better, strong linkage points that have good, but not full, coverage are still useful to link subsets of records. Our framework for linkage point discovery takes input data sets and returns a (ranked) set of linkage points. The discovery is performed in a scalable, online fashion, suitable for large Web data sets, and is illustrated in Figure 2. Task Scheduler, Storage and Indexing Backend. The backend includes a task scheduler that manages (Web) data retrieval, indexing, and discovery tasks, in addition to fast memory and disk-based key-value store, and indexing engines. All the tasks are performed in a way that at any point, partial results can be made available to users of the system. The task scheduler prioritizes tasks and keeps track of their status. Depending on the size of the data, index, and the type of the discovery process, the memory-based key-value store can be used or can be replaced by a disk-based index. Register and Load Data Sets. This component allows users to register a wide variety of data sets. The input could be in XML, RDF (XML, N3 or NTriples), JSON, relational, or CSV format. The data can also come directly from a set of popular Web APIs including those that support SPARQL or the Freebase API. Users can then load data, which will transform their data into our custom JSON format for representing data sets, store it locally, and create basic statistics to help optimize linkage point discovery. Note that our techniques will apply to Web sources that publish data sets of records representing entities of a single type (that is, one data set could be a set of companies, another could be a set of clinical trials, etc.) Furthermore, we make the assumption that each data set represents a set of distinct entities (with few duplicates). This assumption is commonly met by most Web sources which are generally curated. Of course if it is not, we could apply a pre-processing deduplication on each source. Analyze and Index. This component constructs the set of attributes (as defined in Section 2) and indexes the attribute values using one of the following analyzers available in the lexical analyzer library of the system. —Record matching, which identifies the records that represent the same real-world entity, is an important step for data integration. Most state-of-the-art record matching methods are supervised, which requires the user to provide training data. These methods are not applicable for the Web database scenario, where the records to match are query results dynamically generated onthe-fly. Such records are query-dependent and a prelearned method using training examples from previous query results may fail on the results of a new query. To address the problem of record matching in the Web database scenario, we present an unsupervised, online record matching method, UDD, which, for a given query, can effectively identify duplicates from the query result records of multiple Web databases. After removal of the same-source duplicates, the “presumed” nonduplicate records from the same source can be used as training examples alleviating the burden of users having to manually label training examples. Starting from the nonduplicate set, we use two cooperating classifiers, a weighted component similarity summing classifier and an SVM classifier, to iteratively identify duplicates in the query results from multiple Web databases. Experimental results show that UDD works well for the Web database scenario where existing supervised methods do not apply. TODAY, more and more databases that dynamically generate Web pages in response to user queries are available on the Web. These Web databases compose the deep or hidden Web, which is estimated to contain a much larger amount of high quality, usually structured information and to have a faster growth rate than the static Web. Most Web databases are only accessible via a query interface through which users can submit queries. Once a query is received, the Web server will retrieve the corresponding results from the back-end database and return them to the user. To build a system that helps users integrate and, more importantly, compare the query results returned from multiple Web databases, a crucial task is to match the different sources’ records that refer to the same real-world entity. For example, Fig. 1 shows some of the query results returned by two online bookstores, booksamillion.com and abebooks.com, in response to the same query “Harry Potter” over the Title field. It can be seen that the record numbered 3 in Fig. 1a and the third record in Fig. 1b refer to the same book, since they have the same ISBN number although their authors differ somewhat. In comparison, the record numbered 5 in Fig. 1a and the second record in Fig. 1b also refer to the same book if we are interested only in the book title and author.1 The problem of identifying duplicates,2 that is, two (or more) records describing the same entity, has attracted much attention from many research fields, including Databases, Data Mining, Artificial Intelligence, and Natural Language Processing.3 Most previous work4 is based on predefined matching rules hand-coded by domain experts or matching rules learned offline by some learning method from a set of training examples. Such approaches work well in a traditional database environment, where all instances of the target databases can be readily accessed, as long as a set of high-quality representative records can be examined by experts or selected for the user to label. In the Web database scenario, the records to match are highly query-dependent, since they can only be obtained through online queries. Moreover, they are only a partial and biased portion of all the data in the source Web databases. Consequently, hand-coding or offline-learning approaches are not appropriate for two reasons. First, the full data set is not available beforehand, and therefore, good representative data for training are hard to obtain. Second, and most importantly, even if good representative data are found and labeled for learning, the rules learned on the representatives of a full data set may not work well on a partial and biased part of that data set. To illustrate this problem, consider a query for books of a specific author, such as “J. K. Rowling.” Depending on how the Web databases process such a query, all the result records for this query may well have only “J. K. Rowling” as the value for the Author field. In this case, the Author field of these records is ineffective for distinguishing the records that should be matched and those that should not. To reduce the influence of such fields in determining which records should match, their weighting should be adjusted to be much lower than the weighting of other fields or even be zero. However, if a matching rule is learned from representatives of the full data set, then it is highly unlikely that a rule to deal with such fields will be discovered. Moreover, for each new query, depending on the results returned, the field weights should probably change too, which makes supervised-learningbased methods even less applicable. To overcome such problems, we propose a new record matching method Unsupervised Duplicate Detection(UDD) for the specific record matching problem of identifying duplicates among records in query results from multiple Web databases. The key ideas of our method are: 1. We focus on techniques for adjusting the weights of the record fields in calculating the similarity between two records. Two records are considered as duplicates if they are “similar enough” on their fields. As illustrated by the previous example, we believe different fields may need to be assigned different importance weights in an adaptive and dynamic manner. 2. Due to the absence of labeled training examples, we use a sample of universal data consisting of record pairs from different data sources5 as an approximation for a negative training set as well as the record pairs from the same data source. We believe, and our experimental results verify, that doing so is reasonable since the proportion of duplicate records in the universal set is usually much smaller than the proportion of nonduplicates. Employing two classifiers that collaborate in an iterative manner, UDD identifies duplicates as follows: First, each field’s weight is set according to its “relative distance,” i.e., dissimilarity, among records from the approximated negative training set. Then, the first classifier, which utilizes the weights set in the first step, is used to match records from different data sources. Next, with the matched records being a positive set and the nonduplicate records in the negative set, the second classifier further identifies new duplicates. Finally, all the identified duplicates and nonduplicates are used to adjust the field weights set in the first step and a new iteration begins by again employing the first classifier to identify new duplicates. The iteration stops when no new duplicates can be identified. The contributions of this paper include: . To our knowledge, this is the first work that studies and solves the online duplicate detection problem for the Web database scenario where query results are generated on-the-fly. In this scenario, the importance of each individual field needs to be considered, which may vary widely from query to query. This makes existing work based on handcoded rules or offline learning inappropriate. . To our knowledge, this is also the first work that takes advantage of the dissimilarity among records from the same Web database for record matching. Most existing work requires human-labeled training data (positive, negative, or both), which places a heavy burden on users. . A machine learning algorithm is proposed to learn only from an approximated negative training set, which may contain some positive examples, i.e., noise. Most existing work learns from a positive example set that contains no noise. The rest of the paper is organized as follows: In Section 2, related work is reviewed. Section 3 first defines the duplicate record matching problem in the Web database context, and then, presents the UDD record matching method. Section 4 validates the UDD method through experime Most record matching methods adopt a framework that uses two major steps ([17] and [23]): 1. Identifying a similarity function. Using training examples (i.e., manually labeled duplicate and nonduplicate records) and a set of predefined basis similarity measures/functions over numeric and/or string fields, a single composite similarity function over one pair of records, which is a weighted combination (often linear) of the basis functions, is identified by domain experts [20] or learned by a learning method, such as Expectation-Maximization, decision tree, Bayesian network, or SVM ([5], [12], [32], and [35]). 2. Matching records. The composite similarity function is used to calculate the similarity between the candidate pairs and highly similar pairs are matched and identified as referring to the same entity. An important aspect of duplicate detection is to reduce the number of record pair comparisons. Several methods have been proposed for this purpose including standard blocking [21], sorted neighborhood method [20], Bigram Indexing [9], and record clustering ([12] and [27]). Even though these methods differ in how to partition the data set into blocks, they all considerably reduce the number of comparisons by only comparing records from the same block. Since any of these methods can be incorporated into UDD to reduce the number of record pair comparisons, we do not further consider this issue. While most previous record matching work is targeted at matching a single type of record, more recent work ([13], [16], and [22]) has addressed the matching of multiple types of records with rich associations between the records. Even though the matching complexity increases rapidly with the number of record types, these works manage to capture the matching dependencies between multiple record types and utilize such dependencies to improve the matching accuracy of each single record type. Unfortunately, however, the dependencies among multiple record types are not available for many domains. Compared to these previous works, UDD is specifically designed for the Web database scenario where the records to match are of a single type with multiple string fields. These records are heavily query-dependent and are only a partial and biased portion of the entire data, which makes the existing work based on offline learning inappropriate. Moreover, our work focuses on studying and addressing the field weight assignment issue rather than on the similarity measure. In UDD, any similarity measure, or some combination of them, can be easily incorporated. Our work is also related to the classification problem using only a single class of training examples, i.e., either positive or negative, to find data similar to the given class. To date, most single-class classification work has relied on learning from positive and unlabeled data ([14], [15], and [24]). In [25], multiple classification methods are comparedand it is concluded that one-class SVM and neural network methods are comparable and superior to all the other methods. In particular, one-class SVM distinguishes one class of data from another by drawing the class boundary of the provided data’s class in the feature space. However, it requires lots of data to induce the boundary precisely, which makes it liable to overfit or underfit the data and, moreover, it is very vulnerable to noise. The record matching works most closely related to UDD are Christen’s method [8] and PEBL [36]. Using a nearest based approach, Christen first performs a comparison step to generate weight vectors for each pair of records and selects those weight vectors as training examples that, with high likelihood, correspond to either true matches (i.e., pairs with high similarity scores that are used as positive examples) or true nonmatches (i.e., pairs with low similarity scores that are used as negative examples). These training examples are then used in a convergence step to train a classifier (either nearest neighbor or SVM) to label the record pairs not in the training set. Combined, these two steps allow fully automated, unsupervised record pair classification, without the need to know the true match and nonmatch status of the weight vectors produced in the comparison step. PEBL [36] classifies Web pages in two stages by learning from positive examples and unlabeled data. In the mapping stage, a weak classifier, e.g., a rule-based one, is used to get “strong” negative examples from the unlabeled data, which contain none of the frequent features of the positive examples. In the convergence stage, an internal classifier, e.g., SVM, is first trained by the positive examples and the autoidentified negative examples and is then used to iteratively identify new negative examples until it converges. The major differences between Christen’s method, PEBL, and UDD are: 1. The weights used in Christen’s method are static, while in UDD the weights are adjusted dynamically. 2. Both Christen’s method and PEBL use only one classifier during the iterations of the convergence stage, while UDD uses two classifiers that cooperate. When there is only one single classifier in the convergence-stage iterations, the classified results from a previous iteration are used by the same classifier as the retraining examples for the next iteration, which makes it unlikely these results will help the classifier obtain a different hypothesis. Using two classifiers cooperatively can help prevent this problem. PEBL is a general framework for the Web page classification problem and it needs a set of positive training examples, while UDD tackles a slightly different classification problem, online duplicate record detection for multiple Web databases. In this scenario, the assumption that most records from the same data source are nonduplicates usually holds, i.e., negative examples are assumed without human labeling, which helps UDD overcome the training examples requirement. 4. PEBL assumes that the set of positive training examples is correct; whether and by how much its performance will be affected by false positive examples is not known. Our experiments show that UDD’s In this section, the duplicate detection problem in the context of Web databases is first defined in Section 3.1, and then, an overview of our solution to this problem, the UDD method, is presented in Section 3.2. Next, the two main classifiers of UDD are described: a weighted component similarity summing classifier in Section 3.3 and an SVM classifier in Section 3.4. The similarity measure used in our experiments is briefly described in Section 3.5. 3.1 Problem Definition Our focus is on Web databases from the same domain, i.e., Web databases that provide the same type of records in response to user queries. Suppose there are s records in data source A and there are t records in data source B, with each record having a set of fields/attributes. Each of the t records in data source B can potentially be a duplicate of each of the s records in data source A. The goal of duplicate detection is to determine the matching status, i.e., duplicate or nonduplicate, of these s t record pairs. Different users may have different criteria for what constitutes a duplicate even for records within the same domain. For example, in Fig. 1, if the user is only interested in the title and author of a book and does not care about the ISBN information, the records numbered 5 in Fig. 1a and the second record in Fig. 1b are duplicates. Furthermore, the records numbered 5 and 6 in Fig. 1a are also duplicates under this criterion. In contrast, some users may be concerned about the ISBN field besides the title and author fields. For these users, the records numbered 5 and 6 in Fig. 1a and the second record in Fig. 1b are not duplicates. This user preference problem makes supervised duplicate detection methods fail. Since UDD is unsupervised, it does not suffer from this problem. 3.1.2 Assumptions and Observations In this section, we present the assumptions and observations on which UDD is based. First, we make the following two assumptions: 1. A global schema for the specific type of result records is predefined and each database’s individual query result schema has been matched to the global schema (see, for example, the methods in [19] and [30]). 2. Record extractors, i.e., wrappers, are available for each source to extract the result data from HTML pages and insert them into a relational database according to the global schema (see, for example, the methods in [29], [37], and [38]). Besides these two assumptions, we also make use of the following two observations: 1. The records from the same data source usually have the same format. 2. Most duplicates from the same data source can be identified and removed using an exact matching method. Duplicate records exist in the query results of many Web databases, especially when the duplicates are defined based on only some of the fields in a record. Using a straightforward preprocessing step, exact matching, can merge those records that are exactly the same in all relevant matching fields. We investigated 40 Websites for four popular domains on the Web, and as shown in Table 1, found that the simple exact matching step can reduce duplicates by 89 percent, on average. The main reason that exact matching is so effective at reducing duplicates is that the data format for records from the same data source is usually the same for all records. In Table 1, the duplicate ratio is defined as follows: Definition 1. Suppose there are m records extracted from a data source d. There can be n ¼ mðm 1Þ=2 record pairs generated, which are formed by putting every two records together. Suppose t of the n record pairs are duplicate records. The duplicate ratio of the m records then is t/n. 3.1.3 Problem Formulation We formulate the duplicate detection problem following the completion of the exact matching step. We represent a pair of records P12 ¼ fr1; r2g, where r1 and r2 can come from the same or different data sources, as a similarity vector V12 ¼ , in which i represents the ith field similarity between r1 and r2 : 0 i 1: i ¼ 1 means that the ith fields of r1 and r2 are equal and i ¼ 0 means that the ith fields of r1 and r2 are totally different. Note that UDD can employ any similarity function (one or multiple) to calculate the field similarity. The similarity function we use in our experiments is discussed in Section 3.5. We call a similarity vector formed by a duplicate record pair a duplicate vector and a similarity vector formed by a nonduplicate record pair a nonduplicate vector. Initially, two sets of vectors can be built. 1. A nonduplicate vector set N that includes similarity vectors formed by any two different records from the same data source.6 2. A potential duplicate vector set P that includes all similarity vectors formed by any two records from different data sources. Given the nonduplicate vector set N, 7 our goal is to try to identify the set of actual duplicate vectors D from the potential duplicate vector set P. 3.2 UDD Algorithm Overview An intuitive solution to this problem is that we can learn a classifier from N and use the learned classifier to classify P. Although there are several works based on learning from only positive (or negative) examples, to our knowledge all works in the literature assume that the positive (or negative) examples are all correct. However, N may contain a small set of false negative examples. For most general, single-class learning algorithms, such as one-class SVM, these noise examples may have disastrous effects [25]. We propose a method that identifies duplicate vectors in P iteratively, in a way similar to [8] and [36]. However, different from these two works, in which only one classifier is used during the iterations, we employ two classifiers in each iteration that cooperate to identify duplicate vectors from P. Two classifiers are used since we believe that, if there is only one classifier, it is possible that the identified positive instances are not effective enough to retrain the classifier to get a more accurate hypothesis, while two classifiers with different characteristics may discover different sets of positive instances. Thus, the two classifiers can benefit from each other by taking advantage of duplicate vectors identified by the other classifier. The overall UDD algorithm is presented in Fig. 2. In this algorithm, two classifiers in tandem, C1 and C2, identify the duplicate vector set D iteratively. At the very beginning of the algorithm, the negative example set N is used to set the parameters W of C1 (line 2). Then, C1 is used to identify a set of duplicate vectors d1 from P and a set of duplicate vectors f from N (lines 3 and 4 and Fig. 3a). If there are no duplicates at the very beginning, i.e., d1 ¼ , the algorithm will stop at line 6 before any iteration begins. In each iteration, after creating N0 by deleting f from N (line 7) and adding d1 and f into D (line 8), we train C2 using the updated D and N0 (line 9), i.e., using positive and negative examples. Next, we employ the trained C2 to detect new duplicate vectors d2 from P (line 10), and then, remove d2 from P (line 11 and Fig. 3b) and add d2 to D (line 12). After D is updated, D and N0 are used to adjust the parameters W of C1 (line 13) that were set tentatively according to N before any iteration began. Using C1 with the new parameters W, we can identify a new set of duplicate vectors from P (d0 1 in Fig. 3c) and a new set of duplicate vectors from N (f0 in Fig. 3c), and a new duplicate vector detection iteration begins. The iteration stops when no new duplicates are identified by C1 The world-wide web has become the most important information source for most of us. Unfortunately, there is no guarantee for the correctness of information on the web. Moreover, different web sites often provide conflicting information on a subject, such as different specifications for the same product. In this paper we propose a new problem called Veracity, i.e., conformity to truth, which studies how to find true facts from a large amount of conflicting information on many subjects that is provided by various web sites. We design a general framework for the Veracity problem, and invent an algorithm called TruthFinder, which utilizes the relationships between web sites and their information, i.e., a web site is trustworthy if it provides many pieces of true information, and a piece of information is likely to be true if it is provided by many trustworthy web sites. Our experiments show that TruthFinder successfully finds true facts among conflicting information, and identifies trustworthy web sites better than the popular search engines. The world-wide web has become a necessary part of our lives, and might have become the most important information source for most people. Everyday people retrieve all kinds of information from the web. For example, when shopping online, people find product specifications from web sites like Amazon.com or ShopZilla.com. When looking for interesting DVDs, they get information and read movie reviews on web sites such as NetFlix.com or IMDB.com. “Is the world-wide web always trustable?” Unfortunately, the answer is “no”. There is no guarantee for the correctness of information on the web. Even worse, different web sites often provide conflicting information, as shown below. Example 1: Authors of books. We tried to find out who wrote the book “Rapid Contextual Design” (ISBN: 0123540518). We found many different sets of authors from different online bookstores, and we show several of them in Table 1. From the image of the book cover we found that A1 Books provides the most accurate information. In comparison, the information from Powell’s books is incomplete, and that from Lakeside books is incorrect. The trustworthiness problem of the web has been realized by today’s Internet users. According to a survey on credibility of web sites, 54% of Internet users trust news web sites at least most of time, while this ratio is only 26% for web sites that sell products, and is merely 12% for blogs. There have been many studies on ranking web pages according to authority based on hyperlinks, such as AuthorityHub analysis [2], PageRank [4], and more general link-based analysis [1]. But does authority or popularity of web sites lead to accuracy of information? The answer is unfortunately no. For example, according to our experiments the bookstores ranked on top by Google (Barnes & Noble and Powell’s books) contain many errors on book author information, and some small bookstores (e.g., A1 Books) provide more accurate information. In this paper we propose a new problem called Veracity problem, which is formulated as follows: Given a large amount of conflicting information about many objects, which is provided by multiple web sites (or other types of information providers), how to discover the true fact about each object. We use the word “fact” to represent something that is claimed as a fact by some web site, and such a fact can be either true or false. There are often conflicting facts on the web, such as different sets of authors for a book. There are also many web sites, some of which are more trustworthy than some others. A fact is likely to be true if it is provided by trustworthy web sites (especially if by many of them). A web site is trustworthy if most facts it provides are true. Because of this inter-dependency between facts and web sites, we choose an iterative computational method. At each iteration, the probabilities of facts being true and the trustworthiness of web sites are inferred from each other. This iterative procedure is rather different from Authority-Hub analysis [2]. The first difference is in the definitions. The trustworthiness of a web site does not depend on how many facts it provides, but on the accuracy of those facts. Nor can we compute the probability of a fact being true by adding up the trustworthiness of web sites providing it. These lead to non-linearity in computation. Second and more importantly, different facts influence each other. For example, if a web site says a book is written by “Jessamyn Wendell”, and another says “Jessamyn Burns Wendell”, then these two web sites actually support each other although they provide slightly different facts. In summary, we make three major contributions in this paper. First, we formulate the Veracity problem about how to discover true facts from conflicting information. Second, we propose a framework to solve this problem, by defining the trustworthiness of web sites, confidence of facts, and influences between facts. Finally, we propose an algorithm called TruthFinder for identifying true facts using iterative methods. The rest of the paper is organized as follows. We describe the problem in Section 2, and propose the computational model in Section 3. Experimental results are presented in Section 4, and we conclude this study in Section 5. The input of TruthFinder is a large number of facts about properties of a certain type of objects. The facts are provided by many web sites. There are usually multiple conflicting facts from different web sites for each object, and the goal of TruthFinder is to identify the true fact among them. Figure 1 shows a mini example dataset. Each web site provides at most one fact for an object. We first introduce the two most important definitions in this paper, the confidence of facts and the trustworthiness of web sites. Definition 1. (Confidence of facts.) The confidence of a fact f (denoted by s(f)) is the probability of f being correct, according to the best of our knowledge. Definition 2. (Trustworthiness of web sites.) The trustworthiness of a web site w (denoted by t(w)) is the expected confidence of the facts provided by w. Different facts about the same object may be conflicting. However, sometimes facts may be supportive to each other although they are slightly different. For example, one web site claims the author to be “Jennifer Widom” and another one claims “J. Widom”. If one of them is true, the other is also likely to be true. In order to represent such relationships, we propose the concept of implication between facts. The implication from fact f1 to f2, imp(f1 → f2), is f1’s influence on f2’s confidence, i.e., how much f2’s confidence should be increased (or decreased) according to f1’s confidence. It is required that imp(f1 → f2) is a value between −1 and 1. A positive value indicates if f1 is correct, f2 is likely to be correct. While a negative value means if f1 is correct, f2 is likely to be wrong. The details about this will be described in Section 3.1.2. Please notice that the definition of implication is domain specific. When a user uses TruthFinder on a certain domain, he should provide the definition of implication between facts. If in a domain the relationship between two facts is symmetric, and the definition of similarity is available, the user can define imp(f1 → f2) = sim(f1, f2) − base sim, where sim(f1, f2) is the similarity between f1 and f2, and base sim is a threshold for similarity. Based on common sense and our observations on real data, we have four basic heuristics that serve as the bases of our computational model. Heuristic 1: Usually there is only one true fact for a property of an object. Heuristic 2: This true fact appears to be the same or similar on different web sites. Heuristic 3: The false facts on different web sites are less likely to be the same or similar. Heuristic 4: In a certain domain, a web site that provides mostly true facts for many objects will likely provide true facts for other objects. A S many businesses, government agencies and research projects collect increasingly large amounts of data, techniques that allow efficient processing, analysing and mining of such massive databases have in recent years attracted interest from both academia and industry. One task that has been recognised to be of increasing importance in many application domains is the matching of records that relate to the same entities from several databases. Often, information from multiple sources needs to be integrated and combined in order to improve data quality, or to enrich data to facilitate more detailed data analysis. The records to be matched frequently correspond to entities that refer to people, such as clients or customers, patients, employees, tax payers, students, or travellers. The task of record linkage is now commonly used for improving data quality and integrity, to allow re-use of existing data sources for new studies, and to reduce costs and efforts in data acquisition [1]. In the health sector, for example, matched data can contain information that is required to improve health policies, information that traditionally has been collected with time consuming and expensive survey methods [2], [3]. Linked data can also help in health surveillance systems to enrich data that is used for the detection of suspicious patterns, such as outbreaks of contagious diseases. Statistical agencies have employed record linkage for several decades on a routinely basis to link census data for further analysis [4]. Many businesses use dedupli- • Peter Christen is with the Research School of Computer Science, College of Engineering and Computer Science, The Australian National University, Canberra ACT 0200, Australia. E-mail: peter.christen@anu.edu.au cation and record linkage techniques with the aim to deduplicate their databases to improve data quality or compile mailing lists, or to match their data across organisations, for example for collaborative marketing or e-Commerce projects. Many government organisations are now increasingly employing record linkage, for example within and between taxation offices and departments of social security to identify people who register for assistance multiple times, or who work and collect unemployment benefits. Other domains where record linkage is of high interest are fraud and crime detection, as well as national security [5]. Security agencies and crime investigators increasingly rely on the ability to quickly access files for a particular individual under investigation, or crosscheck records from disparate databases, which may help to prevent crimes and terror by early intervention. The problem of finding records that relate to the same entities not only applies to databases that contain information about people. Other types of entities that sometimes need to be matched include records about businesses, consumer products, publications and bibliographic citations, Web pages, Web search results, or genome sequences. In bioinformatics, for example, record linkage techniques can help find genome sequences in large data collections that are similar to a new, unknown sequence. In the field or information retrieval, it is important to remove duplicate documents (such as Web pages and bibliographic citations) in the results returned by search engines, in digital libraries or in automatic text indexing systems [6], [7]. Another application of growing interest is finding and comparing consumer products from different online stores. Because product descriptions are often slightly varying, matching them becomes challenging [8]. In situations where unique entity identifiers (or keys) are available across all the databases to be linked, the problem of matching records at the entity level becomes trivial: a simple database join is all that is required. However, in most cases no such unique identifiers are shared by all databases, and more sophisticated linkage techniques are required. These techniques can be broadly classified into deterministic, probabilistic, and learning based approaches [4], [9], [10]. While statisticians and health researchers commonly name the task of matching records as data or record linkage [11] (the term used in this paper), the computer science and database communities refer to the same process as data or field matching [12], data integration [13], data scrubbing or cleaning [14], [15], data cleansing [16], duplicate detection [17], [18], information integration [19], entity resolution [20], [21], [22], reference reconciliation [23], or as the merge/purge problem [24]. In commercial processing of business mailing lists and customer databases, record linkage is usually seen as a component of ETL (extraction, transformation and loading) tools. Two recent surveys have provided overviews of record linkage and deduplication techniques and challenges [4], [18]. 1.1 The Record Linkage Process Figure 1 outlines the general steps involved in the linking of two databases. Because most real-world data are dirty and contain noisy, incomplete and incorrectly formatted information, a crucial first step in any record linkage or deduplication project is data cleaning and standardisation [25]. It has been recognised that a lack of good quality data can be one of the biggest obstacles to successful record linkage [2]. The main task of data cleaning and standardisation is the conversion of the raw input data into well defined, consistent forms, as well as the resolution of inconsistencies in the way information is represented and encoded [15], [25]. The second step (‘Indexing’) is the topic of this survey, and will be discussed in more detail in Section 2. The indexing step generates pairs of candidate records that are compared in detail in the comparison step using a variety of comparison functions appropriate to the content of the record fields (attributes). Approximate string comparisons, which take (typographical) variations into account, are commonly used on fields that for example contain name and address details [12], while comparison functions specific for date, age, and numerical values are used for fields that contain such data [26]. Several fields are normally compared for each record pair, resulting in a vector that contains the numerical similarity values calculated for that pair. Using these similarity values, the next step in the record linkage process is to classify the compared candidate record pairs into matches, non-matches, and possible matches, depending upon the decision model used [9], [27]. Record pairs that were removed in the indexing step are classified as non-matches without being compared explicitly. The majority of recent research into record linkage has concentrated on improving the classification step, and various classification techniques have been developed. Many of them are based on machine learning approaches [10], [20], [28], [29], [30], [31]. If record pairs are classified into possible matches, a clerical review process is required where these pairs are manually assessed and classified into matches or nonmatches. This is usually a time-consuming, cumbersome and error-prone process, especially when large databases are being linked or deduplicated. Measuring and evaluating the quality and complexity of a record linkage project is a final step in the record linkage process [9]. While various indexing techniques for record linkage and deduplication have been developed in recent years, so far no thorough theoretical or experimental survey of such techniques has been published. Earlier surveys have compared four or less indexing techniques only [32], [33]. It is therefore currently not clear which indexing technique is suitable for what type of data and what kind of record linkage or deduplication application. The aim of this survey is to fill this gap, and provide both researchers and practitioners with information about the characteristics of a variety of indexing techniques, including their scalability to large data sets, and their performance for data with different characteristics. The contributions of this paper are a detailed discussion of six indexing techniques (with a total of twelve variations of them), a theoretical analysis of their complexity, and an empirical evaluation of these techniques within a common framework on a variety of both real and synthetic data sets. The reminder of this paper is structured as follows. In the following Section 2 the indexing step of the record linkage process is discussed in more detail. The six indexing techniques are then presented in Section 3, followed by their experimental evaluation in Section 4. The results of these experiments are discussed in Section 5. An overview of related work is then provided in Section 6, and the paper is concluded in Section 7 with an outlook to future work and challenges in this area. 2 INDEXING FOR RECORD LINKAGE AND DEDUPLICATION When two databases, A and B, are to be matched, potentially each record from A needs to be compared with every record from B, resulting in a maximum number of |A| × |B| comparisons between two records (with | · | denoting the number of records in a database). Similarly, when deduplicating a singe database A, the maximum number of possible comparisons is |A| × (|A| − 1)/2, because each record in A potentially needs to be compared with all other records. The performance bottleneck in a record linkage or deduplication system is usually the expensive detailed comparison of field (attribute) values between records [9], [32], making the na¨ıve approach of comparing all pairs of records not feasible when the databases are large. For example, the matching of two databases with one million records each would result in 1012 (one trillion) possible record pair comparisons. At the same time, assuming there are no duplicate records in the databases to be matched (i.e. one record in A can only be a true match to one record in B and vice versa), then the maximum possible number of true matches will correspond to min(|A|, |B|). Similarly, for a deduplication the number of unique entities (and thus true matches) in a database is always smaller than or equal to the number of records in it. Therefore, while the computational efforts of comparing records increase quadratically as databases are getting larger, the number of potential true matches only increases linearly in the size of the databases. Given this discussion, it is clear that the vast majority of comparisons will be between records that are not matches. The aim of the indexing step is to reduce this large number of potential comparisons by removing as many record pairs as possible that correspond to nonmatches. The traditional record linkage approach [4], [11] has employed an indexing technique commonly called blocking [32], which splits the databases into nonoverlapping blocks, such that only records within each block are compared with each other. A blocking criterion, commonly called a blocking key (the term used in this paper), is either based on a single record field (attribute), or the concatenation of values from several fields. Because real-world data are often dirty and contain variations and errors [34], an important criteria for a good blocking key is that it can group similar values into the same block. What constitutes a ‘similar’ value depends upon the characteristics of the data to be matched. Similarity can refer to similar sounding or similar looking values based on phonetic or character shape characteristics. For strings that contain personal names, for example, phonetic similarity can be obtained by using phonetic encoding functions such as Soundex, NYSIIS or Double-Metaphone [35]. These functions, which are often language or domain specific, are applied when the blocking key values (BKVs) are generated. As an example, Table 1 shows three different blocking keys and the resulting BKVs for four records. The first one is made of Soundex (Sndx) encoded givenname (GiN) values concatenated with full postcode (PC) values, the second consists of the first two digits (Fi2D) of postcode values concatenated with Double-Metaphone (DMe) encoded surname (SurN) values, and the third is made of Soundex encoded suburb name (SubN) values concatenated with the last two digits (La2D) of postcode values. To illustrate the two components of each blocking key in Table 1, their values are separated by a hyphen (‘-’), however in real-world applications they would be concatenated directly. Several important issues need to be considered when record fields are selected to be used as blocking keys. The first issue is that the quality of the values in these fields will influence the quality of the generated candidate record pairs. Ideally, fields containing the fewest errors, variations or missing values should be chosen.