WOO: A Scalable and Multi-tenant Platform for Continuous Knowledge Base Synthesis

Kedar Bellare*
Facebook
kedar.bellare@gmail.com

Carlo Curino*
Microsoft
ccurino@microsoft.com

Ashwin Machanavajihala*
Duke University
ashwin@cs.duke.edu

Peter Mika
Yahoo!
pmika@yahoo-inc.com

Mandar Rahurkar
Yahoo!
rahurkar@gmail.com

Aamod Sane
Yahoo!
aamod@yahoo-inc.com

ABSTRACT

Search, exploration and social experience on the Web has recently undergone tremendous changes with search engines, web portals and social networks offering a different perspective on information discovery and consumption. This new perspective is aimed at capturing user intents, and providing richer and highly connected experiences. The new battleground revolves around technologies for the ingestion, disambiguation and enrichment of entities from a variety of structured and unstructured data sources – we refer to this process as *knowledge base synthesis*. This paper presents the design, implementation and production deployment of the Web Of Objects (WOO) system, a Hadoop-based platform tackling such challenges. WOO has been designed and implemented to enable various products in Yahoo! to synthesize knowledge bases (KBs) of entities relevant to their domains. Currently, the implementation of WOO we describe is used by various Yahoo! properties such as Intonow, Yahoo! Local, Yahoo! Events and Yahoo! Search. This paper highlights: (i) challenges that arise in designing, building and operating a platform that handles multi-domain, multi-version, and multi-tenant disambiguation of web-scale knowledge bases (hundreds of millions of entities), (ii) the architecture and technical solutions we devised, and (iii) an evaluation on real-world production datasets.

1. INTRODUCTION

Traditionally, we have seen three modes of information discovery and consumption – (i) web search has focused on a *document finding* experience, (ii) structured vertical portals focused on search and browsing experiences around specific classes of entities (movies, businesses, products, etc.), and (iii) social networks target communications and interactions between people. These worlds are starting to collide, with large web portals working to create a fully inter-linked graph of entities [8], search companies working to apply such graphs to the problem of *semantic search* [10], and social networks aiming to capture user interests through interactions with real-world entities.

Realizing this ambitious goal requires technologies for the ingestion, disambiguation and enrichment of entities from a variety of structured and unstructured data sources. We refer to this entire process as *knowledge base synthesis*. Over time several web products within Yahoo! have faced similar problems and devised entity extraction, disambiguation and enrichment solutions catering specifically to their domain (e.g., Movies, Locals etc.,) and application needs. In this paper, we describe an ongoing effort to tackle such scenarios in a unified way: the Web of Objects (WOO) initiative—a highly scalable and multi-tenant knowledge base synthesis platform.

The WOO platform is designed to operate on hundreds of millions of entities (and tens of billions of relationship triples) ingested daily from hundreds of semi-structured data feeds and to provide a flexible and extensible platform that can easily support the needs of multiple tenants (the many Yahoo! web properties such as Locals, Movies, Deals, Events, IntoNow), while abstracting out common parts of the solution in a platform layer.

The sheer scale of the data we handle, together with the need to cope with diverse domains, requirements from multiple tenants, the need to support multiple versions of the knowledge bases, and to operate in specific production environment, exposed key challenges we believe are not fully addressed (or maybe even known) today, and yet are of growing relevance within many large web companies.

The list of such problems includes: (1) support for persistent identifiers in face of constant data evolution, (2) support for multiple tenants (with different service level requirements, different data sizes, different semantic interpretation of concepts), (3) handling of skew for very large scale entity matching problems, (4) support for very rich and complex entities, and (5) supporting human curation and algorithmic decisions within the same pipeline.

In this paper, we will summarize (Sec. 2) the business context and the corresponding requirements in a way that should provide valuable insight on the characteristics of the

*Authors in alphabetical order. Additional authors listed in Sec. 13. Work done while at Yahoo! Research

Proceedings of the VLDB Endowment, Vol. 6, No. 11

problems faced at Yahoo! and within our industry.

We then proceed to detail the architecture of our knowledge base synthesis platform (Sec. 3), and present some of the key research challenges we faced in building and operating such a system (Sec. 4-9), such as machine-learning based entity matching at scale, persistent ID management, curation etc. The technical choices we present are the result of careful scientific studies, and pragmatic engineering requirements imposed by our production settings and business needs.

The system we describe is currently running in production, and we thus report as experimental evaluation some of the results we obtain on production datasets (Sec. 10).

Facets of WOO we describe is what we call a "platform-play", where multiple tenants leverage a common infrastructure for knowledge base synthesis in their domain. The next step of our vision foresees the many knowledge basis (KBs) produced by WOO to be further integrated and served by a common infrastructures, in what would be considered a "data-play", with new users accessing directly a shared disambiguated KB.

2. MOTIVATION AND REQUIREMENTS

In this section, (1) we briefly describe the business context within which we have designed, developed and deployed the Web Of Objects platform, (2) we derive some of the key requirements, and (3) we introduce some of our broad technological decisions. This will frame the rest of the presentation and help understand how challenges we face influence our design decisions and imposed priorities in the development of the platform itself.

2.1 Business Context

Yahoo! is one of the largest web companies in the world, and through many years of organic growth and acquisitions has become a complex business reality comprised of many sites including our popular *Search* engine, and several verticals such as *Locals*, *Events*, *Sports*, *News*, and many others.

Over the years each of the web properties of Yahoo! has independently attempted to solve the core problem of building "high quality" knowledge bases extracting content from disparate and noisy data sources (paid data feeds, user generated data, web crawl, etc.) The WOO project represents a *unified approach* to tackle this problem systematically across the entire company, and as such, it is subject to requirements from multiple business units—we will refer to such units as customers or tenants of our platform.

The ultimate goal of most of the platform customers is to provide to Yahoo's over 700 million users serendipitous access to fresh, rich and authoritative content.

Desiderata. The underlying knowledge base should satisfy the following properties:

P1 *Coverage*: This measures the fraction of real-world entities included in the knowledge base (KB). Higher coverage means that even lesser known (tail) entities are included in the KB. This is particularly critical for scenarios where users care dearly about the long-tail: e.g., local businesses.

P2 *Accuracy*: The information in the knowledge base must be accurate. For instance, a wrong home page or a incorrect phone number associated with a restaurant can be frustrating for a user.

P3 *Linkage*: This describes the level of connectivity between entities. Higher linkage means richer navigational experiences across entities. For instance, a knowledge base of movies and actors is more useful to users if actors are linked to the movies they acted in.

P4 *Identifiability*: One and only one identifier (e.g., URL) should be consistently used to refer to a unique real-world entity. This is perceived by the user as a form of coherency of the KB with the real world, and it is one of the key problems we tackle.

P5 *Persistence / Content Continuity*: Entities in the real world change over time – actors change their marital relationships, new restaurants open in place of old ones that close, etc. A key requirement is variants of the same entity across time must be linked, preferably by the same identifier (e.g., URL), and their content should reflect the lifecycle of the entity in the real world.

P6 *Multi-tenant*: The KB that is constructed must be useful to multiple portals. Multiple tenants might have different freshness and quality requirements, care about a different aspect of the KB, and even have varying interpretations of common entity types.

Challenges. The features above are deemed to be crucial by our customers in order to drive user engagement and creating coherent user experiences. Meeting these requirements helps us frame the key challenges we need to address. Specifically, we need to *scale* in three important dimensions – *size*, *domains*, and *across time*. Recent work [9] has shown that constructing a KB of entities and their attributes with high coverage requires integrating information from a number of sources (specifically, tens of thousands of sites). This means handling:

C1 *Large Data size*: handling graphs of hundreds of millions of entities with hundreds of attributes for each entity.

C2 *Heterogenous Input Formats/Schemas*: importing data from highly heterogeneous input formats and schemas

C3 *Diverse Data Quality*: adapting to both high quality data sources (e.g., clean feeds from advertisers) and noisy ones (e.g., user generated content or content extracted from text and html pages).

Next, the multi-tenancy and linkage requirements motivate the need to build solutions that can easily work across multiple domains. This creates the following challenges:

C4 *Multiple Domains*: allowing to easily customize the solution to handle new domains, and cross-domains operations.

C5 *Heterogenous Output Formats/Schemas*: allowing customizable export functionalities is vital to cope with a complex business reality.

Finally, the real world changes over time. Therefore, we need to ensure that the knowledge base we synthesize reflects the life cycle of the data in the real world. Hence, we need to cope with:

C6 *History-aware KBs*: ensuring that historical versions of the entities in the knowledge base are used for synthesis and are exposed to the users.

C7 *Persistence*: ensures that different temporal versions of the same entity are linked (preferably using the same identifier). This is an important challenge, especially in a Web portal setting, where entities in a knowledge base are exposed to users. These users associate information (search queries, tags, photos, reviews, bookmarks etc.), which we call *eyeballs*, and persistence ensures that these eyeballs are correctly associated with the right entities despite changes to the entities in the real world.

C8 *Incremental Maintenance*: is required instead of bulk processing when the number of updates to the real world are continuous and need to be reflected in real-time.

In summary, we are interested in *"scalable, multi-tenant, and continuous knowledge base synthesis"*, the process of continuously extracting, disambiguating, curating and consistently maintaining massive collections of linked entities extracted from potentially noisy and heterogeneous data sources on behalf of multiple tenants. This problem is closely related to a series of classic problems such as data integration and data exchange [23], entity de-duplication [11], data archival and schema evolution [19], yet it introduces certain novel aspects such as scalability, multi-tenancy, and temporal continuity or persistence.

2.2 Design Decisions

In the rest of this section, we briefly discuss main design decisions and how they were driven from the requirements and challenges mentioned previously. The rest of the paper dives more deeply in the architecture and describes in detail system design and building.

Entity Resolution on Hadoop. Many of the data sets we encounter at Yahoo! contain hundreds of millions of entities with hundreds of attributes. Entity resolution at such a scale fundamentally requires distributed computation framework. In the future, we expect our system to be applied to larger corpora with billions or hundreds of billions of entities. Therefore, we leverage the access to a internal massive shared Hadoop infrastructure for performing large-scale distributed computations. Our approach is similar to many of the recently proposed approaches to large-scale entity resolution using MapReduce [14, 18].

Plugin-based Architecture for Multi-tenancy. Many of our challenges (especially, C4 and C5) arise due to the fact that our knowledge base is used to synthesize entities from many different domains. WOO is designed as a *platform* which allows for domain level customization as required by our internal products. Our platform handles the overall process flow and scaling of algorithms while allowing clients to customize certain modules by way of *plugins*. Customers can focus on implementing their own plugins to handle nuances of their data without getting distracted by scaling for grid which is managed by the platform. This design eases the deployment and evolution of the platform by decoupling customer and platform code bases by means of clear interfaces.

Batch and Incremental Processing of Updates. Our system must handle updates to the entities in the knowledge base (typically impacting a few thousand entities a day) under stringent service level agreements (SLAs). The combination of high quality matching and the need to support stringent SLAs force us to provide both a batch pipeline, and a mechanism to incrementally integrate changes, called *Fastpath*. The latter allows rapid entity resolution leveraging the output of the batch deduplication as a reference KB, thus satisfying the SLAs. However, this incremental resolution can result in errors since it does not have a global view of the data. Such errors are fixed by periodically running the batch de-duplication algorithm.

Scalable Editorial Judgments Tools. Our pipeline makes two type of algorithmic decisions: (a) matching, that is, which objects to merge because they are duplicates, and (b) content continuity, which is how to assign persistent ids to object clusters across time. Since our algorithms are statistical, they can make mistakes in these decisions. Hence, we need a way that editors can override both of these decisions. For these reasons, we provide a mechanism for the editors to explicitly say whether two objects should be merged or not. It also provides a way of tying certain objects to persistent IDs. One approach is to add speciality override hooks in different system components, but this adds even more complexity to an already complex system. Instead, we annotate source objects with additional attributes such as *must-merge-with-target-objects*, *should-not-merge-with-target-objects* and *desired-persistent-ID*. These annotations are then used as very strong evidence while deciding to merge or split entities and during assignment of persistent IDs.

3. ARCHITECTURE OVERVIEW

In this section, we describe the system architecture, the overall data flow in the system and briefly introduce the design of each functional component.

Figure 1 provides a pictorial representation of the architecture of our platform. Boxes with round corners represent software components (yellow boxes are used to distinguish customizable parts), arrows represent data flows and are labelled using (document-like) rectangles describing the type of data exchanged between software components. Colored backgrounds are used to group of software components in functional subsystems or "phases" of our pipeline.

Each software component depicted here constitutes a set of map-reduce jobs, written in PIG with generic UDFs, that run over Hadoop. With support from the WOO team the platform is customized to match the needs of each internal customer, and is run for such customer on a shared grid. Resulting KBs are shared (and integrated) across customers whenever relevant, and leveraged to create a unified KB.

Input / Output. The input of the platform is a collection of structured or semi-structured data sources. Their data formats can include XML feeds, RDF content, Relational Databases, or other custom formats. The output is a fully integrated and de-duplicated KB, both in a format consistent with the WOO data model/schema and in any custom format required by the platform customers.

Importer. In the first step of our pipeline the importer, whose aim is to provide uniformity, converts each data source to a common format called the WOO schema. The abstract data model of our system is conceptually equivalent to RDF

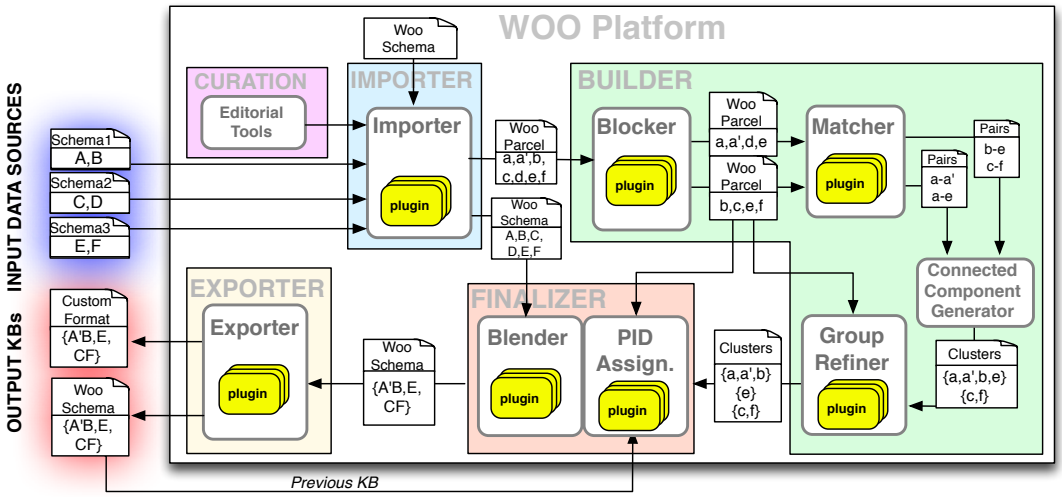


Figure 1: The WOO Architecture

as further detailed in Section 4.1, but its physical representation is optimized for efficiency. This includes native compression, and de-normalization to reduce random seeks during processing. The importer is customizable and can be easily extended to ingest new data sources. In order to reduce I/O requirements of the many intermediate stages of our pipeline the importer provides, together with the full representation of the rich input entities (often hundreds of attributes), a more compact representation – containing only the attribute strictly needed for matching – called a WOO Parcel. As Figure 1 shows, only the WOO Parcel is pushed through the pipeline, while the complete entities are used only in the later stages.

Builder. The next phase performs the actual entity deduplication.

- *Blocker.* Since we operate in the hundreds of millions of input entities it is not practical to perform N^2 comparisons to test matching of each pair of input entities. Therefore, to reduce the number of needed comparisons we exploit a (user-defined) blocking function aimed at bucketing the data in a way that does not harm recall, yet heavily reduce the complexity of the downstream operators.
- *Matcher.* In this step, a combination of machine-learned and rule-based algorithms are used to decide whether a pair of entities match or not. Customers of our platform can either leverage library of matching algorithms included in the platform or supply their own matching plugin. The output of this phase is a set of pairs of entities together with their matching scores.
- *Connected Components.* Next connected components are computed to generate "clusters" of entities that are likely to be matching. This is accomplished by leveraging the Giraph bulk synchronous graph-processing system. This procedure computes the transitive closure of the matching defined above in a distributed fashion.
- *Group Refiner.* This optional stage can further refine potentially large clusters created during connected components. More collective approaches suggested in previous literature can be applied at this stage. The output of this stage is a clustering of the entities.

Blender. At this point the full entities (we now operate not on a WOO Parcel but on the entire source data) are blended according to a user defined function. A simple scenario is taking the union of the values for each attribute, but more complex functions are used to increase quality and respect contractual agreements with our source providers (e.g., paid feeds have priority over crawled data, and editorial curated content has an even higher priority).

Persistent ID Assignment. The previous steps produce a clustering decision based on the source object content, which we call an "equivalence over space." On the other hand, as the object content as well as the matching algorithm evolve over time, the clustering decision might change. Hence, it is necessary to maintain the correspondence over successive versions of the clusters to make sure that identity continuity is consistent with the content continuity. We call this an "equivalence over time." These two dimensions of equivalences capture the life cycle of a real world entity. This step assigns persistent identifiers for objects in the KB while making an effort to keep identifiers as stable as possible over time. We note that identifier stability in spite of evolving data was a major request from internal customers.

Exporter. The final step is the Exporter component, which allows the customer of our platform to retrieve the overall KB in any output format. Several default exporters ship with the platform (targeting the key serving platforms within Yahoo!, and standard formats such as RDF).

Curation. In order to enable editorial curation of the content we provide a set of GUIs that enable domain experts to influence the system behavior, for example, by forcing or disallowing certain matches between entities, or by editing attribute values. These judgements flow into the system as if they are an additional input data sources. However, they are handled with particular care by the system (e.g., as higher priority inputs overriding some of the algorithmic decisions). By treating editorial judgments as yet another data source, editorial and algorithmic decisions are handled uniformly, and this allows for potential mistakes in the editorial judgements. As an example, during the group refiner stage we can detect and resolve conflicts among editorial suggestions, such as disagreement between editors on whether two input entities are duplicates or not. Editorials tools influencing

the input and custom plugins for some of the stages of the pipeline enable for complete editorial control.

Two important functionalities not shown in the figure are *FastPath*, i.e., an incremental version of this pipeline that allows fast modifications to the KB (i.e. additions, deletions, updates), and a *Serving* layer, that enables users to browse and search KBs.

The rest of the paper provides further details of each software component introduced above.

4. INPUT/OUTPUT

Our data model builds on graph-based representations of knowledge as commonly used in the field of the Semantic Web. Graph-based data models are a natural choice for capturing knowledge of real world entities, their attributes and their relationships. More specifically, they allow a fine-grained representation of knowledge at the level of individual facts (statements of attribute values or relationships) that are necessary for integration tasks.

Our data model is an extension of RDF with additional metadata beyond the built-in *datatype* and *language* attributes for literals. It allows provenance tracking of individual facts (both attributes and relationships) by keeping metadata about which publisher produced a particular assertion and in what context. Recall that this is necessary because an individual entity may accrue information from a variety of sources and we would like to keep track of the source of individual assertions. This is helpful in tracking lineage and many other reasons, foremost among which is the support for fine-grained content licensing.

We assign namespace-based UUIDs to entities. It is up to the consumers of the data to assign one or more human-readable aliases and publishing URIs to entities, e.g. to map the UUID of the entity corresponding to the movie "Red Dawn" to the user- and SEO-friendly URI `http://movies.yahoo.com/movie/red-dawn-2010-2/`.

This abstract data model is represented on disk using an internal format called the Common Content Model. CCM is a JSON-based document format with a schema language (Common Content Schema, CCS) similar to Avro and an efficient binary serialization. Each document captures information about a single entity.

4.1 Schema

Formal schemas (ontologies) are a key component of our integration platform even though our data model could capture data without a schema. Different actors in our system may have different conceptualizations: even within a single domain (e.g. local business listings) different publishers would use slightly different structure and semantics (e.g. restaurants vs. eating establishments) when publishing data. Likewise, individual consumers have their own target representations. With n publishers and m consumers, without a shared schema we would potentially need to maintain $n \times m$ mappings between input and output schemas. Shared schemas also protect the system and consumers of the data from changes in the source schemas. This is particularly important for reusing rules and editorially created training data. Our system also has the future goal of supporting an integrated knowledge base when there are overlaps in datasets across domains e.g. venues of events in an event data set may also appear as locations in a Local dataset. Our schema captures such connections across domains.

The WOO schema is formally represented using the Web Ontology Language (OWL), a powerful schema language supported by multiple tools (e.g., Protégé). Unlike with entities, we assign HTTP URIs to all schema elements and publish the documentation on our intranet. From a practical perspective, this means that an engineer inspecting the data can copy the URI of a class or property to retrieve a machine- or human-readable definition. Before publishing we check that the ontology is consistent and look for common editorial mistakes such as missing labels or descriptions. We use OWLDiff to compute semantic diffs between different versions of the schema, effectively auto-generating a changelog of schema changes. Since OWL is based on Description Logic, we are also able to determine when schema changes have been introduced that are potentially backward incompatible. Last, OWL allows data validation and inference. We can perform limited validation by translating the OWL definition of the classes to CCS. This has obvious limitations in that only local constraints can be checked. In a separate work, we transform SPARQL queries to PigLatin, and thus validate the SPARQL axiomatization of the ontology [17].

5. IMPORTER

The importer is a data feed specific component that normalizes the input data to abstract the input schema connotations and changes from the subsequent processing of the entities. This tackles the well understood problem of schema mapping and data exchange [12].

However, solving this problem in a production setting was more challenging than expected. Many of the existing tools only handle relational or XML data models, and only tackle specific types of mappings. The class of schema matching/exchange we tackle calls for rather generic functionalities (support for arbitrary input data models, support for complex structural matching, support for attribute-value manipulations). This forced us to take a pragmatic approach and tackle this problem manually. Ongoing work is devoted to automate this phase as well.

The current solution requires our customer to export their data in the CCM format mentioned above. Next we provide platform support for individual customers to build personalized importers that map their private schemas into the shared WOO schema. Platform users write XSLT and Javascript when needed. This choice was very well received by our internal customers since they were already familiar with XSLT. The Hadoop based design is very scalable during the data exchange phase once the mapping is manually defined.

The importer is broken into multiple steps:

Schema Normalization: performs arbitrarily complex mappings (many-to-many attributes and entity mappings are allowed) that are specified by the user.

Attribute Normalization: is executed using a series of user specified heuristics (e.g, for address attributes "St." strings are converted into "Street").

Validation: checks entities and attributes according to a set of business and domain rules (e.g., February has 28-29 days).

Parcel Creation: ensures that the data format is amenable to high-performance execution. The KB that adheres to a highly normalized ontological schema is represented in a

heavily de-normalized fashion (e.g., movies will "contain" all the actors that acting in the movie), and we project out (temporarily) all the attributes not useful throughout the pipeline for entity reduplication. This heavily contributes to performance by reducing data volumes and making most data accesses sequential.

While each of the above steps is currently manually devised by the tenants of our platform we provide significant infrastructural support to help the customization (e.g., the denormalization step is expressed declaratively by the users, while the platform provides scalable transitive-closure logic that transforms the data according to the user specification).

The output of the importer is a WOO schema-compliant KB that is generated in both a full and a projected format (i.e., a WOO Parcel) and used throughout the platform.

6. BUILDER

This section describes the *builder* component of the pipeline which handles entity de-duplication and normalization. It operates on the WOO parcels created by the importer. Each entity type has its own builder (e.g. *BUSINESS*, *VENUE*, *PERFORMER*) and multiple builders can run simultaneously or consecutively in the platform. The steps involved in the builder as described earlier in Section 3 include: (i) blocking, (ii) matching, (iii) connected component generation and (iv) refining. We describe the default behavior for many of these plug-ins below.

6.1 Blocking

As discussed earlier in Section 2.2 entity de-duplication is based on pairwise comparison of KB entities. This can be computationally expensive for large KBs since $O(n^2)$ comparisons are needed for a KB with n entities, while proper blocking reduces the complexity to $O(Bm^2)$ where B is the total number of blocks and m is the user-configurable maximum block size with small or no impact on recall (more details in our experimental section). Users control the blocking function, where single entities are allowed to appear in multiple blocks (for recall). The blocking plug-in takes as input the WOO parcel and outputs a set of block identifiers or hashes. Entities with the same identifier or hash are then grouped together. Blocks that contain a single entity or too many entities (e.g., block size > 1000) are ignored. It is desirable that hash function(s) used in the plug-in capture some notion of similarity between two entities. Many similarity functions are known to have corresponding hash functions that approximate them. In separate work [22], we have also built an automatic blocking mechanism for learning hash functions that maximize recall while minimizing number of pairwise comparison required in the downstream phases. At the end of blocking, the blocks produced are given to the pairwise matching module.

6.2 Pairwise Matching

The next step in the process compares all pairs of entities within a block and outputs whether an entity pair matches or not. Optionally, a real-valued score can also be produced for each pair to indicate the strength of the match. Firstly, given an entity pair a *matcher* computes a set of real-valued features using feature functions. These feature functions can range from generic similarity metrics to more domain-specific ones. Next, these real-valued features are combined using a rule-based, machine-learned or a hybrid model. Many common feature functions (e.g., Jaccard similarity, Edit distance, etc.) and machine learning models (e.g., AdaBoost, Logistic Regression and Random Forests) are implemented in a library written specifically to handle WOO parcels. A machine-learned model also requires a "golden" data set (GDS) consisting of matching and non-matching entity pairs. Such a data set is curated manually and frequently updated based on errors observed after running the pipeline (more details in Section 6.5 below). Finally, the result of applying the model to each entity pair (i.e., score and boolean match decision) are passed on to the next stage in the pipeline.

6.3 Connected Components (CC)

This phase connects matching pairs distributed across blocks since an entity can be placed in multiple buckets due to hashing. For example, entities A and B can be in the block because they share a common phone number and entities A and C in block because they have identical names. After running CC, we may find that $\{A, B, C\}$ are in one component and also that they are part of a single entity. CC is provided as part of the platform and uses *Giraph* (a *Pregel*-like parallel graph processing system) to accomplish this task at scale.

6.4 Refining

In the final phase of the builder, the goal of the refiner is to fine-tune the entity connected components, if needed, to produce entity clusters. We occasionally observe that CC produces very large components that consist of multiple entity clusters. Refining provides users the opportunity to refine or break such components into more sensible entity clusters. As with previous algorithms, this step is also plug-in based. The plug-in takes as input a single connected component, the pairwise match decisions and match scores and outputs a set of entity clusters. The user may choose to return the connected component itself as a cluster, return cliques in the component as entity clusters or adopt a middle ground that accounts for errors made by the matching algorithm.

One common plug-in used in this step is based on a correlation clustering algorithm by Ailon et. al. [1]. This algorithm represents the middle ground as stated above and is robust to errors made during matching. In correlation clustering, we are given a graph with $\{+, -\}$ edges and the objective is to find a clustering that minimizes the number of $-$ edges within clusters and $+$ edges between clusters. Here the $+$ and $-$ edges correspond to match and non-match decisions between entity pairs. Although this problem is NP-hard in general, we leverage the randomized algorithm described in [1]. It randomly picks a vertex and creates a cluster out of the vertices connected to it with $+$ edges. It then recurses on the remaining graph (i.e. after removing previously clustered vertices). We have observed that this algorithm is both fast and produces reasonably good entity clusters. In practice, we find that it is beneficial to run the algorithm multiple times with different random seeds and then return the clustering with the highest matching score. The score of a clustering is given by the difference of matching scores within clusters and those between clusters. It also corresponds to the log-probability of a clustering if a logis-

tic regression model is used during matching. An additional benefit of this algorithm is that it can easily incorporate both soft and hard rules provided by editors and additional business rules regarding matches.

6.5 Expanding to new domains

The employment of plug-in based architecture makes it easy to expand the entire system to add new domains from an engineering point of view. However, many data characteristics and match definitions may vary from one domain to the next. The engineering harness is in place to support faster pace evolution, but much research is still needed to automate such process. To respond to new domains, and to guarantee quality is maintained over time a serious framework for quality evaluation is required. In the rest of this subsection we summarize the creation, maintenance of "golden data sets" (i.e., manually curated data set we use for quality evaluation) and the related scalability challenges.

6.5.1 Golden Data Set

When a new domain is provided to us by a customer, the first step involved is exploring the data characteristics such as the number of input sources, size of the sources, quality of attributes in each source (e.g. manually curated or automatically extracted) and other relevant statistics. In addition, we also require the definition of what it means for entities to match along with specific use cases to help during editorial judgments.

Often a key challenge in learning a new match model is the absence of a "golden data set" (GDS) – a labeled dataset for training the matcher models. A naive way of obtaining such a dataset for our matcher is to randomly select pairs from the input sources, and manually judge them to check whether a pair of records refer to the same entity. However, when the size of the input ranges in millions, it is highly likely that a sample of size even in hundreds would yield pairs that are mostly mismatches, and there would not be enough labeled data to train a matcher. Instead, we bootstrap our GDS using a heuristic blocker and matcher that uses simple blocking functions (to have almost 100 percent recall), and fast matching to hunt for probable matches. We use an approach similar to the one proposed by [4] in which a heuristic matcher outputs four kinds of pairs: (i) exact matches, (ii) high-confidence heuristic which has high precision but low recall of matches, (iii) low-confidence heuristic which has low precision but high recall of matches and lastly, (iv) non-matches which are randomly sampled pairs from the input sources. We are careful to include example pairs of different qualities and ensure that they cover all or many of the input sources. These pairs are then judged by editors based on the provided guidelines. These judgments are used to bootstrap a machine-learned matching model. Once a match model has been bootstrapped, we can rapidly iterate by sampling new pairs for judgment from the data on which the machine-learned model is uncertain.

The process described above is facilitated by editorial tools that we integrated in the builder. The editorial tool has two phases: (i) a local operation phase in which the editor continuously refines the blocking and matching specifications (e.g. weights, features, thresholds, etc.), and, (ii) a grid (HDFS) phase in which the current specifications of blocking and matching are used to sample new examples from datasets on HDFS for the local phase.

We also explored active learning for entity de-duplication [3]. One simple technique for active learning is based on the observation that examples should be sampled proportional to their uncertainty. This technique is implemented in the grid phase of the editorial tool by sampling more examples that are closer to the match threshold.

6.5.2 Evaluation

Operating at scale also makes the evaluation of the builder precision and recall challenging.

We first evaluate the grid output after each stage (i.e. blocking, matching, CC, refining) on the current GDS and ensure that pairwise precision and recall are reasonable and meet standards proposed by the customer. By construction the GDS contains cases that are most likely "hard" and both machine and humans would find confusing. Such pairs, however, constitute a minor fraction of all the matching pairs since a majority of the matching pairs are easy to detect. Hence, we also adjust for counts of high vs low confidence matches in actual data while reporting the final quality of the builder. An example of this evaluation for local businesses is provided in Table 3 (see Section 10).

We also use other approaches to evaluate the precision and recall of the builder output. To evaluate precision, we sample record pairs (e.g. 100 pairs) from clusters in the final builder output and present them to the editors for judgments. The proportion of matching pairs constitutes an unbiased estimate of the precision of our matcher. If this does not meet the required standards then further iterations of GDS pipeline are executed. Similarly, for recall, we load the builder output into a search index and ask editors to search for a variety of businesses. If they find duplicates in the search results (e.g. top 20 results) these are reported as recall errors. Again, if the recall across all queries does not meet the requirements it triggers further iterations of GDS.

7. FINALIZER

The finalizer is the last stage of the KB building pipeline. This stage is responsible for handling the *persistence* of object identifiers and the blending of the attributes of the (potentially many) entities that are being merged.

7.1 Persistency Problem

Persistence is one of the requirements we received from our internal customers, and captures the implicit expectation from human users that a real-world entity will be perpetually represented by a single *persistent identifier* (PID). While natural evolution of the content (such as the update of a phone number, or change in address) is acceptable, there is a strong expectation of stability for identifiers. For instance, having a persistent identifier allows maintaining stable URLs for objects in a Web portal. This very reasonable requirement is actually rather hard to achieve in the face of evolving input data and evolving matching logic.

Another motivation for PIDs is the problem of correctly maintaining the association between output objects and user generated annotations – e.g., photos, reviews, ratings, clicks, searches and tags. These annotations, which we call *eyeballs*, constitute a very important form of enrichment for an output object. Over time source objects change leading to new entities (a new build of the KB, newKB), and eyeballs from old entities (previous build of KB, prevKB) must be correctly associated with new entities.

EXAMPLE 1. *Consider a scenario where the* prevKB *has a cluster P of the following supposedly duplicate listings – (i) [ABC Cafe, 10 Main St, CA], (ii) [AB Cafe, 10 Main St, San Francisco, CA], and (iii) [AB Cafe, 10 Main Street, San Francisco, CA]. Suppose, users have associated 5 photos and 120 reviews with entity P.*

Now, if we get an update to listing (i) as [ABC Cafe, 10 Main St, Berkeley, CA], then this gives evidence that AB Cafe and ABC Cafe are indeed two distinct entities in different cities. Hence, newKB *may choose to split P into two – C_1 containing [ABC Cafe, 10 Main St, Berkeley, CA], and C_2 containing (ii) [AB Cafe, 10 Main St, San Francisco, CA], and (iii) [AB Cafe, 10 Main Street, San Francisco, CA]. We need to now decide how to split the eyeballs (photos and reviews) across the two new entities.*

Persistent identifiers provide a simple solution for carrying over eyeballs from old to new entities. In the above example, if C_2 is given the same PID as P, then all eyeballs associated with P are associated with C_2.

Formally, the persistent identifier assignment problem is defined as following: In the successive builds B_{old} and B_{new}, let S_{old} and S_{new} denote the source object sets respectively, which share some common source objects, but also have exclusive ones. In the preceding build B_{old}, the source objects are clustered and persisted as $P_{old} = \{o_i\}_{i=1}^{p}$, in which o_i denotes the persisted cluster id, which contains source objects from S_{old}, while in the current build B_{new}, the clustered result is $C_{new} = \{n_j\}_{j=1}^{q}$, in which n_j denotes the cluster id which contains source objects from S_{new}. The persistent identifier assignment problem is to find an optimal cluster mapping M from P_{old} to C_{new}, such that if o_i and n_j are mapped according to M, then the share the same persistent ID and most likely must represent the same real world entity.

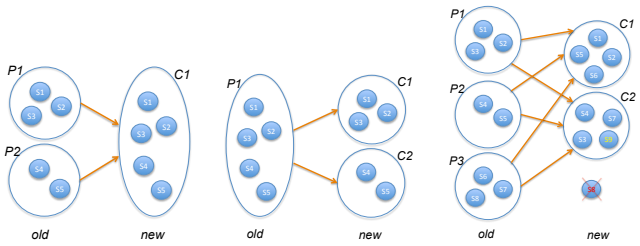
7.2 Source Persistence Assumption

Conceptually, computing the optimal cluster mapping across builds is identical to the record linkage problem across two databases. This can be done by (a) computing the pairwise similarity between every pair, and (b) finding a bipartite matching that maximizes the total similarity. However, the complexity of the naive pairwise comparison algorithm is $O(N^2)$. Instead, WOO makes the following assumptions to ensure efficiency:

- *SID Continuity*: If two source objects (from the same feed) have the same identifier (but with different content), then they correspond to information about the same real world entity.
- *Surjection*: Each source object is mapped to one (and only one) persisted object

Based on the above id assumptions, the source identifier could be regarded as a strong signal about the object content. For instance, if we take the SID as proxy for the content, we can infer that if two content clusters have the same SIDs, they are considered stable. Therefore a full content comparison is only applied for clusters that have different SIDs. The cluster comparisons are categorized as follows:

1. *Identical*: both old and new cluster contain the same source objects;
2. *Delete*: all the source objects of the old cluster are marked as deleted;



(a) Merge (b) Split (c) Shuffle

Figure 2: Patterns: Merge, Split, and Shuffle

3. *New*: all the source objects of the new cluster are observed for the first time in the new build;
4. *Merge*: all the information from source objects in multiple old clusters are merged into a single new cluster;
5. *Split*: all the information from source objects in the old cluster are spread across multiple new clusters;
6. *Shuffle*: mixture of merge and split patterns;

Figure 2 illustrates some examples of the patterns described above. In the figure, source objects are denoted as atomic ball tagged with $s*$, deleted objects are crossed out in red, and new objects are highlighted in yellow.

7.3 Persistence Algorithm

Persistence pattern identification step uses the source id as a bridge to separate the clusters into the aforementioned persistence patterns. In a gradually evolving knowledge base, the majority of entities do not change at all, as a result, the *Identical* pattern covers most of the entities. *Merge*, *Split* and *Shuffle* patterns are handled in a unified way and can be treated as a single *Connected* pattern.

Mapping Table Construction: In the case of *Identical, New, Deleted* patterns the mapping is obvious. For *Connected* pattern the problem is transformed into a bipartite graph matching problem, where the disjoint sets represent the old and new clusters. First, we calculate the pairwise cluster similarity, and filter out those pairs with low scores. We then generate direct mapping pairs using a greedy bipartite graph matching algorithm, these pairs belong to *Mapped* category and are directly persisted. For those old clusters which are not in the direct mapping pairs, we find the best matching new cluster and redirect the old cluster to the new one. If there is no such best matching new cluster found, we mark the old cluster as (soft) deleted. Finally, for the new clusters which are not in the direct mapping pairs, we assign new persistent identifiers.

7.4 Blending

A final step in the pipeline is *blending*, or the problem of computing *canonical* attribute values for the entity representing an output cluster. WOO allows user-defined plugins to construct a canonical attribute value either using simple heuristics (e.g., longest string for names, or union), using voting, or by computing a centroid value [6]. WOO also maintains the historical versions of each source object, and thus we have also developed techniques for computing the canonical values that leverage these historical attribute values (described in separate work [20]).

8. EXPORTER

The very last step of our pipeline is the exporter, which is responsible for the final data transformation, and output of the KBs. Once again this component is plug-in based and allows for customizations based on business or domain knowledge, it is implemented as a MapReduce job, where the input is a full scan of the KB.

The second step of the exporter allows arbitrary transformations of data formats, fundamental to support the disparate needs of our internal customers. While the transformation logic is typically provided by the users, the platform supports dereferencing of entity references, where the user can select which attributes to de-normalize on and over how many hops. This simplifies the user code and allows it to operate over arbitrary "neighborhoods". This is convenient when the output format should be heavily denormalized for serving purposes (e.g., our export of actors for the IntoNow application had to contain a list of movies and tv-shows they appeared in).

The WOO system ships with few built-in exporters: RDF, HBase/HFile, Vespa (Yahoo's internal search index), MG4J (open source IR toolkit).

9. MORE SUPPORT FOR MULTI-TENANCY

To further support multi-tenancy in our system we provide two more useful features: (1) FastPath, i.e., an incremental fast version of our pipeline, and (2) general purpose serving infrastructure.

FastPath. While batch oriented computation like the one we discussed so far provides more stable results (since the algorithms have a consistent global view of the data), they are not compatible with one important use case imposed by our customers: *low-latency integrations of small deltas*.

To address this need we built a *FastPath* component in the WOO platform which simulates an incremental matching approach to provide consistent, millisecond level response times while trying to make the same matching decisions that the batch system would.

FastPath is built using a search engine to identify matching records, based on a simplifying assumption that a new record introduced via the FastPath would not alter the merge and split decisions already made by the builder. That means a new record will either (1) merge with an existing entity cluster, or (2) form a new entity cluster.

In the FastPath, we index each WOO entity by its blocking keys using the same blocking functions as the batch build process, and store the corresponding WOO parcel for each entity. To identify matches for a *query record*, the search engine applies the same importer schema transformations and blocking functions to the record. The resulting blocking keys form a query which is executed against the search index to identify *candidate* matching entities. A candidate entity is more likely to be a good match for the new record if it shares several (yet less common) blocking keys, which allow us to use fairly standard techniques in the inverted index to rank the candidates and inspect only a small subset of the possible matches.

The top k candidates are ranked and then compared against the query record using the retrieved WOO parcels and the same pairwise comparison functions of the batch system. If a candidate has a sufficiently high similarity score, it is returned as the match. Otherwise, the query record is assigned a new identifier and inserted into the FastPath index as a new entity. The decisions made by the FastPath are then passed into the batch system to maintain continuity in subsequent runs, as the batch build may later override the decision made by the FastPath. As each batch run completes, the exporter process generates a set of updates for the FastPath index. Many more details on the FastPath implementation are discussed in [26].

General purpose serving. Another common requirement we observed emerging was the one of a general purpose browsing and querying interface. While our current tenants tend to heavily customize the exporting and serving of the KBs, developers, current tenants, editors, and future tenants have the frequent need to access the content of this large KBs through some general purpose querying and browsing interface that promotes a serendipitous exploration.

The key requirements for a general purpose browsing system are: (1) storing and serving tens of large KBs (equivalent to hundreds of billions of triples), (2) fast bulk loading (minutes for multi-billion triples KBs), and (3) provide fast point lookups, browsing, and structured queries.

After considering several existing RDF endpoint we decided to roll our own HBase + MG4J solution. To this purpose we leverage the HBase + MG4J exporter, that builds in parallel HBase index files (bulk-loading functionality of HBase), and MG4J indexes—for more details on the RDF indexing see [5]. The resulting indexes are loaded in a cluster of 40 nodes, supporting loading times in the order of tens of minutes for KBs in the billions of triples and fast serving. A HTML5 browsing and querying interface facilitates unstructured interactions with the data.

10. EXPERIMENTS AND EVALUATION

We ran our experiments on a typical Hadoop/Linux grid with 3009 map slots and 1672 reduce slots, with 2GB main memory heap per slot.

We present results on two datasets for entity matching: EVENTS and LOCALS. These are production datasets that power web portals in Yahoo!, namely `beta.local.yahoo.com` and `local.yahoo.com`. Each of these datasets come with their own set of challenges. LOCALS is the larger of the two datasets with over 100 million input business listings coming from more than 50 different sources (feeds), where each source object has over 50 attributes. EVENTS dataset contains 4 types of source objects (events, venues, performers and occurrences) with references across these objects. Details of these datasets are presented in Table 1.

Metric	Events	Local
# source objects	> 200 K	> 100 M
# input feeds	30+	50+
# entity types	4	1

Table 1: Datasets in production.

10.1 Brute force matching

If we were to resort to brute force ($O(n^2)$) matching, where we compare each pair of source objects, for LOCALS dataset that would need $> 10^{16}$ comparisons. Our best pairwise matching algorithm (as we will see later) takes 450μs per comparison. Thus, it would take more than 142,000

years on a single machine, and over 45 years even on our largest grids.

Metric	Events	Local
Precision	0.98	0.95
Recall	0.91	0.92
# of M/R Jobs	68	68
Batch run time	5 *hrs*	24 *hrs*
# of blocks containing more than 1 object (blocking)	19,950	4×10^7
Parallel Reducers	400	800
Total Mappers	98,111	245,433
Total Reducers	6,001	30,812
Map heap size	4GB	4GB
Reduce heap size	4GB	4GB
Giraph Mappers	100	200
Giraph heap size	2GB	4GB

Table 2: Overall Results

10.2 Overall WOO Pipeline Results

Table 2 shows the overall performance numbers for the WOO system on our two datasets. The entire WOO workflow, from the import to the finalize stage, took 24 hours in the case of LOCALS and 5 hours in the case of EVENTS. The table also provides information about the number of map-reduce tasks in the workflow, and the maximum number of mappers and reducers across all the tasks.

As described in Section 6.5.2, we evaluate the result of the WOO workflow using the following adjusted precision-recall measure. Note that the GDS was constructed by picking samples from four strata – exact matches, high/low confidence matches, and obvious non-matches, which were then editorially evaluated (Sec. 6.5.1) . Precision is computed as the fraction of pairs in the GDS that are identified as matches and are truly a match. Recall is the fraction of matches in the GDS that are correctly identified as matches. The adjusted precision and recall are computed by accounting for the sampling rate from each of the 4 strata. Table 2 shows that the adjusted precision and recall are quite high for both datasets.

10.3 Characteristics of the Builder

Events dataset required 4 bootstrap iterations with final sample size of 1500 sample points while Local dataset needed 15 bootstrap iterations with final sample size of 6K data points. Table 3 illustrates the precision and recall (on a fixed held-out GDS) after each of the steps in the builder workflow for the LOCALS dataset. The last row shows the adjusted precision and recall of the final output. All other rows report the actual precision/recall on the GDS. After blocking, less than 2% of the pairs which are labeled as matches do not occur in any block together. The *initial bootstrap* of matching uses a GDS constructed by a standard methodology [4] and consists of editorially judged pairs that have been randomly sampled and those matching high-confidence and low-confidence heuristics (see Section 6.5). The GDS is

Builder Stage	Precision	Recall	F1
Blocking	–	0.981	–
Matching (initial bootstrap)	0.895	0.812	0.851
Matching (final)	0.876	0.923	0.899
Connected Components	0.842	0.970	0.901
Refining	0.900	0.886	0.893
Adjusted	0.952	0.923	0.937

Table 3: Quality measures after each stage of the builder for LOCALS dataset.

Method	LOCALS	EVENTS
Logistic	450 μs	220 μs
AdaBoost	610 μs	390 μs
Random Forests	860 μs	420 μs

Table 4: Average time taken to compute the probability of match for one pair of records.

updated further to include pairs that the matcher is uncertain about and also based on error analysis after precision and recall evaluations. This improves the F1 of the *final* matcher to almost 90%.

Table 4 shows the average time take to complete one pairwise matching operation on the LOCALS dataset for three different classifiers – Logistic Regression, AdaBoost and Random Forests. Figures 3(a) and 3(b) show the precision and recall (on the GDS) of the three classifiers on LOCALS and EVENTS datasets, respectively. While there is no significant difference in the performance of the three classifiers, logistic is the fastest, and we use this in the production system.

10.4 PID Assignment

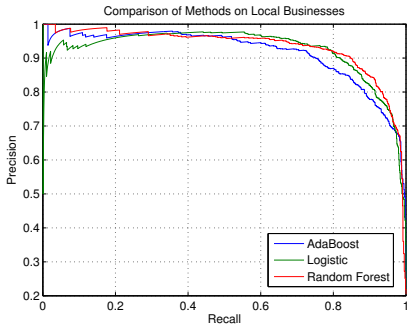
Persistent Identifier (PID) assignment is an integral part of the finalize stage and acts as a key feature of WOO system. We ran our persistence algorithm on a dataset with over a million events and venues. We observed that across two builds (separated by a week), less than 1% of the entities changed their IDs. Quantifying the accuracy of this algorithm requires constructing a test set with labels identifying whether pairs of entities from the old and new KBs correspond to the same real-world object. While we do not have such a test set at this time, we have qualitative feedback from our customers that IDs are indeed correctly persisted.

We also implemented an alternate approach to computing persistent IDs that does not make the source persistence assumption. We compare pairs of clusters that were blocked together by a hash-based blocking scheme. However, this scheme resulted in only 30% of the entities retaining the same ID across the two builds. This could be explained by two reasons – (a) hash based blocking does not perform very well with clusters of source objects, and (b) blocking does not work well, because values of source objects change over time, and source IDs are a better signal for persistence. A detailed analysis of persistence is ongoing.

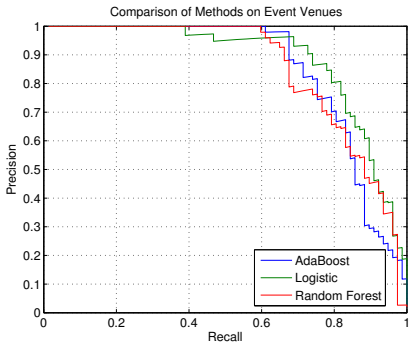
10.5 FastPath Processing Results

Due to space constraints, we summarize the results using an experiment which measures the effectiveness of the FastPath making match decisions for previously unseen records. A more complete evaluation of the incremental features of our system is available in [26].

We use the EVENTS dataset for this experiment and first construct a knowledge base using the batch processing method. We then uniformly at random sample 1% of the EVENTS input records into a subset $S_{1\%}$. For each record O_i in $S_{1\%}$, we identify the cluster in the knowledge base which contains O_i as a source and remove all attribute values, relations, and blocking keys provided to that entity by O_i. We create an inverted index from this modified knowledge base and use the records in $S_{1\%}$ to query the FastPath. This means that, for evaluation purposes, attribute values from those excluded records are not present in the index for candidate retrieval or in the WOO parcels for pairwise matching.



(a)



(b)

Figure 3: (a) and (b) compare various machine learning pairwise matchers over LOCALS **and** EVENTS **datasets respectively.**

We evaluate the precision, recall, and F-measure for retrieving the top $1 \leq k \leq 10$ candidate entities using the search engine's default text ranking, using the batch result as the ground truth for comparison. We consider the returned result a positive match if the query record was matched with the suggested cluster from the full knowledge base. The other possibilities are false positive matches (matching the incorrect cluster), false negative matches (returning no result), or true negatives (if the excluded record was the only source record for a particular cluster).

Figures 4 and 5 show the precision, recall, F1 and running time of introducing new records to the system via the FastPath. Figure 4 shows the precision ranges from 0.904 to 0.896 as we retrieve up to 10 candidate entities, while at the same time recall improves from 0.951 to 0.979. The average running time for a query is approximately $20ms$ on a single commodity machine.

Error type	k=1	k=5	k=10
Incorrectly resolved	1.19%	1.31%	1.34%
Incorrectly non-resolved	0.57%	0.31%	0.25%

Table 5: New record misclassification rates

The results show that for applications which prefer aggressive de-duplication at the potential cost of incorrectly merging records, retrieving more candidate entities is preferable, while applications which require more certainty in de-

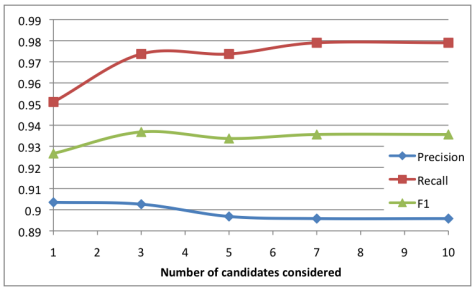


Figure 4: Top-n retrieval performance for new records

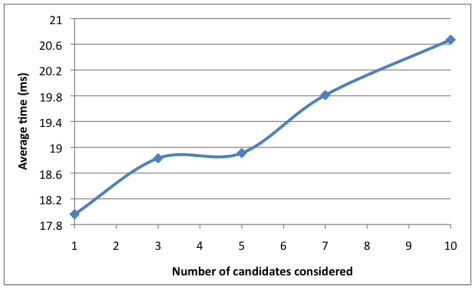


Figure 5: Query time new records

duplication decisions can opt to retrieve fewer candidate entities. Table 5 shows the breakdown of misclassification rates for $k = 1$, $k = 5$, and $k = 10$, highlighting this tradeoff. In either scenario, the FastPath achieves approximately 90% precision while also finding over 95% of the matching entities in the knowledge base.

11. RELATED WORK

Entity de-duplication/disambiguation/resolution have received increased research attention in recent years. Surveys of entity resolution frameworks and the related notions of data-fusion appeared in: [16, 15]. Parallel blocking and entity resolution have been studied in [7, 14, 18, 24], and have been explored in the context of noisy and heterogeneous data in [21]. Recently [14] presented an Hadoop-based entity de-duplication framework, that shares some of the underlying goals with WOO, and leverage a similar cluster infrastructure.

All of the above systems, however, mainly tackle the problem of scalable entity resolution in a batch setting. To the best of our knowledge, no single system has addressed the problem of *"scalable, multi-tenant, continuous knowledge synthesis"* in its entirety. Our system has the following advantages:

- *Plugin-based Architecture*: Our system has a good separation between the platform components (e.g. ingestion of feeds, blocking, matching, persistent ID assignment, etc.) and the application-specific logic (e.g. matching functions, editorial judgments, etc.). This allows tenants to experiment with various algorithms (e.g. rule-based vs machine-learned matching) and also ramp up quickly on new domains they are interested in, while the platform focuses on ensuring SLAs and scaling computation.

- *Incremental and Batch Processing*: Previous approaches have mostly focused on batch processing where entity resolution is performed on the corpus as a whole. However, many properties at Yahoo! require incremental processing when there are thousands or hundreds of thousands of new entities streaming in that require resolution. The FastPath approach [25] is transparent to the customer and handles this problem by leveraging existing domain-specific plugins for blocking and matching.
- *Knowledge Base Versioning and Persistent IDs*: In our system, both entities and the algorithms to synthesize the KB change over time. Our system transparently handles the persistence of identifiers for the different temporal versions of the same entity. We also store historical versions of the KB efficiently so that entities and algorithm decisions can be tracked over time.
- *Editorial Judgments*: The notion of editorial judgments is built into the system. Such judgments are used in combination with statistical and rule-based approaches at various stages of our pipeline to create an accurate KB. We believe that the combination of editorial judgments and algorithmic decisions are crucial to the construction of accurate knowledge bases. Unlike previous approaches, we handle such judgments transparently as a core input to the system.

We are aware of at least three competing efforts within other large companies (Facebook, Google, and Microsoft [2, 13]). We suspect that these companies are tackling similar problems, and have built comparable solutions. While no direct comparison is possible due to the lack of information, we believe the existence of such systems testifies the growing relevance of this topic and the clear economical potential of efficient entity deduplication at scale.

12. CONCLUSIONS

We presented the Web Of Objects, a platform that tackles Scalable, Multi-tenant and Continuous Knowledge Synthesis. We detailed the key design decisions and provide an overview of our multi-tenant plugin-based architecture. Throughout the paper we highlighted novel aspects of known problems, or new problems we found along the way while building and deploying such system. We then introduced our approach to solve such problems, motivating technical decisions based on research effort and business constraints. While comparable systems exist today within other large organizations, at the best of our knowledge our paper is the first to discuss the internals of a production-grade entity deduplication framework. We experimentally evaluated the system on production datasets, and demonstrate high precision and recall and unparalleled scalability. We also reported on the usability and extensibility of the system when faced with multi-tenant scenarios. Finally we articulated the challenges and requirements for a system handling continuous data deduplication, and presented our approach to tackle this new problem.

13. ADDITIONAL AUTHORS

We list additional authors in alphabetical order: Phil Bohannon, Laukik Chitnis, Chris Drome, Anup Goyal, Zhiwei Gu, Ashok Halambi, Balaji Kannan, Vibhor Rastogi, Parin Shah, Nicolas Torzec, Michael Welch. We would also like to thank Gaurav Mishra, Irfan Mohammad, Ralph Rabbat, and Vijaykumar Rajendrarao for their efforts in ensuring the success of the project.

14. REFERENCES

[1] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *J. ACM*, 2008.
[2] A. Arasu, C. Ré, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *ICDE*, 2009.
[3] K. Bellare, S. Iyengar, A. Parameswaran, and V. Rastogi. Active sampling for entity matching. In *KDD*, 2012.
[4] M. Bilenko and R. J. Mooney. On evaluation and training-set construction for duplicate detection. In *KDD*, 2003.
[5] R. Blanco, P. Mika, and S. Vigna. Effective and efficient entity search in rdf data. In *ISWC*, 2011.
[6] A. Culotta, M. Wick, R. Hall, M. Marzilli, and A. McCallum. Canonicalization of database records using adaptive similarity measures. In *KDD*, 2007.
[7] G. Dal Bianco, R. Galante, and C. A. Heuser. A fast approach for parallel deduplication on multicore processors. In *SACC*, 2011.
[8] N. Dalvi, R. Kumar, B. Pang, R. Ramakrishnan, A. Tomkins, P. Bohannon, S. Keerthi, and S. Merugu. A web of concepts. In *PODS*, 2009.
[9] N. Dalvi, A. Machanavajjhala, and B. Pang. An analysis of structured data on the web. *VLDB*, 2012.
[10] A. Efrati. Google gives search a refresh. *Wall Street Journal*, 2012.
[11] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *TKDE*, 2007.
[12] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Trans. Database Syst.*, 2005.
[13] J. Gemmell, B. Rubinstein, and A. K. Chandra. Improving entity resolution with global constraints. *CoRR*, 2011.
[14] L. Kolb, A. Thor, and E. Rahm. Dedoop: Efficient Deduplication with Hadoop. In *VLDB*, 2012.
[15] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 2010.
[16] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 2010.
[17] S. Kotoulas, J. Urbani, P. Boncz, and P. Mika. Robust runtime optimization and skew-resistant execution of analytical sparql queries on pig. In *ISWC*, 2012.
[18] B. McNeill, H. Kardes, and A. Borthwick. Dynamic Record Blocking: Efficient Linking of Massive Databases in MapReduce. In *QDB*, 2012.
[19] H. J. Moon, C. Curino, M. Ham, and C. Zaniolo. Prima: archiving and querying historical data with evolving schemas. In *SIGMOD*, 2009.
[20] A. Pal, V. Rastogi, A. Machanavajjhala, and P. Bohannon. Information integration over time in unreliable and uncertain environments. In *WWW*, 2012.
[21] G. Papadakis and W. Nejdl. Efficient entity resolution methods for heterogeneous information spaces. In *ICDE Workshops*, 2011.
[22] A. D. Sarma, A. Jain, A. Machanavajjhala, and P. Bohannon. An automatic blocking mechanism for large-scale de-duplication tasks. In *CIKM*, 2012.
[23] B. ten Cate and P. G. Kolaitis. Structural characterizations of schema-mapping languages. *Commun. ACM*, 53(1), 2010.
[24] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, 2010.
[25] M. J. Welch, C. Drome, and A. Sane. High quality real-time incremental entity resolution in a knowledge base.
[26] M. J. Welch, C. Drome, and A. Sane. Fast and accurate incremental entity resolution relative to a batch resolved corpus. In *CIKM*, 2012.