Building Ranked Mashups of Unstructured Sources with Uncertain Information

Mohamed A. Soliman
University of Waterloo
m2ali@cs.uwaterloo.ca

Ihab F. Ilyas
University of Waterloo
ilyas@uwaterloo.ca

Mina Saleeb
University of Waterloo
msaleeb@cs.uwaterloo.ca

ABSTRACT

Mashups are situational applications that join multiple sources to better meet the information needs of Web users. Web sources can be huge databases behind query interfaces, which triggers the need of ranking mashup results based on some user preferences.

We present MashRank, a mashup authoring and processing system building on concepts from rank-aware processing, probabilistic databases, and information extraction to enable ranked mashups of (unstructured) sources with uncertain ranking attributes. MashRank is based on new semantics, formulations and processing techniques to handle uncertain preference scores, represented as intervals enclosing possible score values.

MashRank integrates information extraction with query processing by asynchronously pushing extracted data on-the-fly into pipelined rank-aware query plans, and using ranking early-out requirements to limit extraction cost. To the best of our knowledge, both the technical problems and target applications of MashRank have not been addressed before.

1. INTRODUCTION

The Web is today's most convenient medium for finding information. Web search has evolved from simple keyword lookup to advanced search that integrates data from multiple sources on demand. The recent proliferation of ad-hoc mashup tools [1, 2, 3, 20] allows Web users to build data integration applications with data coming from multiple sources. Due to Web interactive and dynamic nature, building mashups at Web scale triggers the need to rank large volumes of constantly-changing search results with respect to some preference measures. Ranking acts as an effective and intuitive data exploration tool in this scenario.

Traditionally, ranking queries compute the top-k query results based on a given scoring function. A join query augmented with ranking specifications (i.e., a scoring function and a parameter k), usually referred to as "rank join query" [12, 13, 17], reports the top-k join results based on the computed scores. The central idea of rank join is to allow for early query termination by making use of sorted inputs and scoring function monotonicity to upper bound the scores of non-materialized join results.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

1.1 Motivation and Challenges

Consider a Web user planning to spend a vacation in New Zealand. The user would like to find a hotel with a good reputation and reasonable prices. For credibility, The user has decided to obtain hotel pricing and rating information from two independent sources: `www.vianet.travel`, a source that provides pricing information and online booking, and `www.tvtrip.com`, a source that provides hotel ratings from travelers' reviews. The search scenario is thus (1) **extracting** hotel records from both sources; (2) **matching** records based on hotel name; and (3) **ranking** matched results based on some function of *price* and *rating*. We refer to such process as *extract-match-rank*.

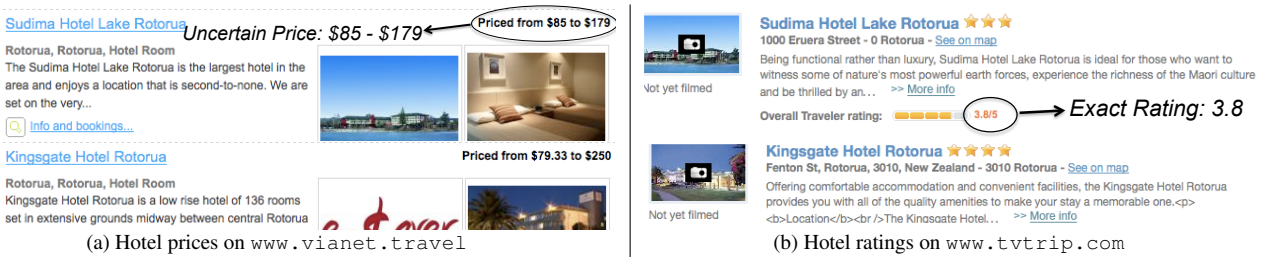
Figure 1 shows actual snapshots of the two sources. Manual processing of the *extract-match-rank* task requires navigating through many pages, while memorizing and ranking interesting matches, which is clearly infeasible. The problem is far more complicated if matching across more sources is involved. An automated *extract-match-rank* system can probably formulate the task in the following SQL-like rank join query, which prefers hotels with low prices and high ratings, and reports only the top-k hotels.

```
SELECT *
FROM vianet, tvtrip
WHERE vianet.HotelName ∼ tvtrip.HotelName
ORDER BY 500-vianet.Price+ 100* tvtrip.Rating
LIMIT k
```

However, such formulation raises challenges on multiple levels:

1. *Data Extraction.* Web sources often lack schema and attribute annotations, since their contents are mainly given as unstructured HTML. Structured records need to be extracted from such sources to apply rank join techniques.
2. *Interleaving Extraction with Processing.* Users do not tolerate long waiting times, where expensive extraction must complete before query processing starts. Moreover, exhaustive extraction does not leverage early-out nature of ranking. We thus need to interleave asynchronous data retrieval and extraction with query processing, and avoid unnecessary extraction operations based on ranking requirements.
3. *Handling Uncertainty.* Web data can contain missing/inexact values (e.g., in Figure 1(a), *price* is a range). This can be attributed to privacy concerns, (e.g., exact price is only known when initiating a purchase transaction), and presentation formats (e.g., a range of prices for all room types of a hotel). When such uncertainty impacts queried/ranked attributes, query processing becomes problematic. Mashing up arbitrary sources further complicates the problem by magnifying uncertainty effect as more sources are joined.

While many studies address the first challenge (extraction) from various perspectives (e.g., supervised/unsupervised learning [8,



(a) Hotel prices on `www.vianet.travel` (b) Hotel ratings on `www.tvtrip.com`

Figure 1: Hotel search mashup

14]), a limited attention has been given to interleaving extraction with rank-aware query processing, as well as handling extracted results with uncertainty. In this paper, we aim at integrating data extraction with rank join processing under uncertainty.

Interleaving Extraction with Processing. Many Web sources provide interfaces for sorted data access. For example, `vianet.travel` in Figure 1(a) provides hotel records ordered on highest prices by adding a parameter *sort_by=highest_price* to the URL. This allows pushing sorted records on-the-fly into mashup execution. Moreover, by avoiding the need to sort extracted records, extraction can be done on demand, which limits extraction cost based on ranking requirements (e.g., do not extract records that will get low scores).

Handling Score Uncertainty. In Figure 1, applying the scoring function given in the query produces the following join results (we show how to compute score ranges in Section 4.2):

Hotel	Price	Rating	Score
Sudima Hotel ...	\$85 - \$179	3.8	701 - 795
Kingsgate Hotel ...	\$79.33 - \$250	3.7	620 - 790.67

The scoring models adopted by current rank join techniques assume that all scores are exact, which yields a unique top-k answer (score ties are resolved using a tie-breaking criterion). In situations where the underlying data do not conform to these assumptions, similar to the results above, the semantics and processing techniques of current rank join methods become inapplicable.

Possible approaches to rank-join multiple inputs with uncertain scores include the following:

(1) Fall back to exact scores. For example, we use expected score values. While suitable in some settings, this approach can be quite unreliable due to the inability to reflect variance of score ranges. Exact score representation of multiple ranges may coincide, or become very close to each other, even though ranges are considerably different. For example, assume 3 records, t_1, t_2, and t_3 with uniform score ranges $[0, 100]$, $[40, 60]$, and $[30, 70]$, respectively. All expected scores are equal to 50, and hence all orderings are equally likely. However, based on how score distributions overlap, the likelihood of different orderings can be computed as nested integrals [22], which results in different probabilities: $\Pr(\langle t_1, t_2, t_3\rangle) = .25$, $\Pr(\langle t_1, t_3, t_2\rangle) = .2$, $\Pr(\langle t_2, t_1, t_3\rangle) = .05$, $\Pr(\langle t_2, t_3, t_1\rangle) = .2$, $\Pr(\langle t_3, t_1, t_2\rangle) = .05$, and $\Pr(\langle t_3, t_2, t_1\rangle) = .25$. That is, some orderings are more likely than the others, even though score ranges are uniform with equal expectations.

(2) Compute all, then rank. The problem of ranking with uncertain scores coming from a single input has been addressed in [22]. However, when scores are computed online (e.g., by aggregating scores of joined Web sources), applying the techniques in [22] requires computing all uncertain scores before ranking. Moreover, [22] assumes the independence of the random variables representing tuples' scores, which does not apply to join results' scores that are intrinsically correlated. We give more details in Section 4.4.

Other Applications. While our main driving application is mashups, we believe that our study is relevant to other contexts as well. For example, in sensor networks [10], sensor readings can be represented as intervals with associated density functions due to the inability to continuously maintain the latest readings. An example rank join query is to find the locations with the best temperature and light settings by joining streams of sensor readings based on location. In location tracking [5, 6], the distance between two moving objects is modeled as an interval enclosing the accurate distance based on location history. An example rank join query is to find the closest pairs of similar moving objects by joining the moving objects based on their types. Finally in information aggregation (e.g., airfare aggregators), the values of aggregated attributes are often modeled as sets/ranges of possible values. An example rank join query is to join the output of an airfare aggregator service with hotel costs in destination cities to find the cheapest vacation packages.

1.2 Contributions

We introduce MashRank[†], a system that provides an integrated solution addressing the aforementioned challenges, and allowing for novel use of mashup and ranking tools for unstructured uncertain data. Our key contributions are summarized as follows:

- We give an architecture for a new mashup authoring and processing system leveraging early-out capabilities of ranking, and integrating on demand extraction with relational query processing (Section 3).
- We introduce the first formulation of rank join under uncertainty, and give new query definitions that can be adopted in various application scenarios (Sections 4.1 and 4.2).
- We extend rank join methods to handle uncertain scores, and provide a pipelined query operator implementation of uncertainty-aware rank join algorithm. The implementation can be integrated into relational query plans (Section 4.3).
- We present a new infrastructure for probabilistic rank join based on Monte-Carlo simulation. The infrastructure handles correlations among join results using a novel join-aware sampling method, and incrementally rank joins results under multiple probabilistic ranking semantics (Section 4.4).

We also conduct an experimental study on real data to evaluate the scalability and efficiency of our methods (Section 5).

2. BACKGROUND AND RELATED WORK

Rank join methods [12, 13, 17, 18] build on using sorted inputs to incrementally report ranked join results by bounding the scores of non-materialized join results. The proposed techniques mainly differ in the maintained state of partial joins (i.e., joins that may lead to valid join results), which can either be a lightweight state that gives loose score bounds (e.g.,[12]), or a dense state, of all partial joins, that gives tight score bounds (e.g.,[18]). The scoring

[†]Prototype is accessible at: http://prefex.cs.uwaterloo.ca/MashRank/.

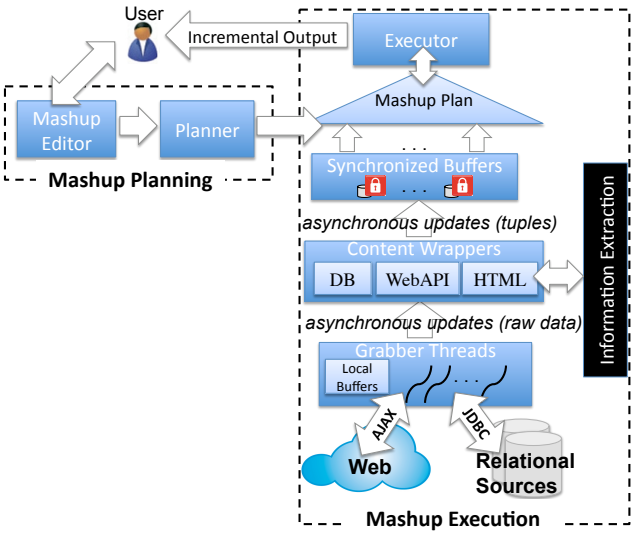


Figure 2: MashRank architecture

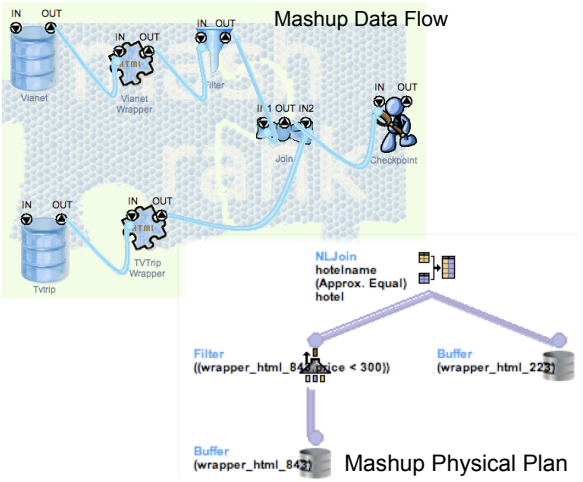


Figure 3: A Screenshot from MashRank prototype

model in all of these methods is deterministic (i.e., each record has a single score), and hence they cannot be applied to settings with uncertain scores. We discuss in Sections 4.3 and 4.4 how to extend rank join algorithms to handle uncertain scores, and integrate joining with probabilistic ranking.

Information extraction techniques approach unstructured data from different perspectives. Supervised learning methods (e.g., [4, 14]), learn extraction rules from a set of user-specified examples by generalizing common properties in these examples. On the other hand, unsupervised learning methods (e.g.,[8, 19]) focus on learning a grammar/template describing the schema of the underlying source by exploiting repeated structure and domain knowledge. The learned template can be used to populate relational tables out of the unstructured sources. MashRank provides a mashup authoring tool (cf. Section 3) that builds on supervised extraction methods, namely wrapper induction. We allow users to annotate and refine examples, during mashup data flow creation, and use these examples to learn extraction rules. We elaborate on our adaptation of wrapper induction in Appendix D.

Current mashup systems (e.g.,[1, 2, 3, 20]) allow creating data flows involving services, sources, and operators. Most systems assume data in the form of pre-computed structured feeds, with the exception of [21], which integrates text extractors into enterprise mashups. However, ranking is mostly overlooked in these works by generating non-optimized plans (e.g., materialze-sort plans) for ranked mashups. Moreover, although uncertainty is ubiquitous on the Web (e.g., missing/inexact values), current systems do not allow querying/reasoning about such uncertainty. Our work integrates concepts from information extraction, rank-aware processing, and probabilistic databases domains to address these problems.

3. SYSTEM ARCHITECTURE

We describe the details of different components in MashRank architecture (given in Figure 2).

Mashup Editor builds a mashup data flow, by interacting with the user, to identify source schemas, join/filter conditions, and scoring function. A mashup data flow is a tree whose leaves are the sources, and internal nodes are three primary logical operators: extractors, joins, and filters†. The edges between tree nodes are pipes indicating the flow of tuples from one logical operator to its parent.

Mashup Planner maps the mashup data flow into a rank-aware and uncertainty-aware physical plan. A mashup physical plan needs to be rank-aware if the user provides a scoring function to order mashup results. The physical plan needs to be uncertainty-aware if the scoring function involves at least one uncertain attribute (e.g. *price* in Figure 1(a)).

Mashup Planner exploits sorting capabilities of input sources to offload sort to source side. For example, if the scoring function involves an attribute that has sorted access (as provided by its corresponding source), the created mashup plan pushes source records directly (i.e., without a sorting phase) into rank-aware mashup execution. This also allows limiting extraction on pages that are not required to compute the top-ranked results (by upper bounding the scores of records not yet extracted).

Figure 3 shows a mashup data flow, constructed using MashRank editor, and a corresponding nested-loops join physical plan. We elaborate on planning and sort offloading in Appendix E.

Content Wrappers bridge different data models to the relational model. Mashups may involve data sources of different models (e.g., XML generated by Web API's, raw HTML, and relational data). MashRank adopts a simple relational model in which mashup (intermediate) results are represented as tuples.

Wrapping HTML into relational tuples is complicated by the lacking of schema information (Challenge 1 in Section 1.1). We build HTML wrappers based on the concepts of *wrapper induction* [14] in information extraction. The idea is to request user to provide a number of labeled examples of different attributes in a sample of source pages. A wrapper inductor learns a rule that correctly extracts all of the given examples. For example, the rule can be maximal strings in HTML source that delimit all of the examples. Applying the rule to other source pages, with the same structure of labeled pages, produces the required tuples. We elaborate on wrapper induction details in Appendix D

Grabber Threads grab data from mashup sources. When sources are accessible through URLs, MashRank initiates a thread for each URL to grab contents. Each thread forwards source response, once ready, to the appropriate wrapper. This allows MashRank to process multiple requests in parallel, and avoid getting blocked on slow sources. When sources are relational, a database connection is used to retrieve tuples from remote database.

Mashup Executor executes the physical plan generated by the Planner against live data sources, and incrementally reports

†Our framework is extensible, which allows other logical operators such as union and intersection.

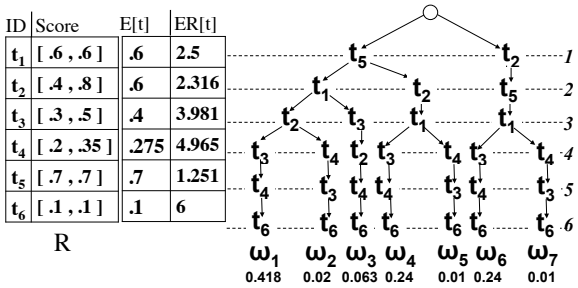


Figure 4: Orderings space for tuples with uniform scores

mashup results to the user. We build on the iterator model (OPEN-GETNEXT-CLOSE), used in most DBMSs, for mashup execution. OPENING the root operator in a query plan tree recursively initializes all tree operators. Processing the query is done by calling the GETNEXT method of the root operator repeatedly until it returns an empty result. Finally, CLOSING the root operator recursively shuts down all operators in the tree.

However, in contrast to relational plans that read data from tables with known sizes residing on disks, a mashup plan may read data from remote sources of unknown sizes filling up asynchronously as more tuples are extracted. Hence, mashup execution needs to be (1) *synchronized*: to guarantee correct reads/updates, sources are locked when wrappers attempt to append new extracted tuples, or when parent nodes in the mashup plan attempt to read next tuple; and (2) *push-based*: an operator requesting tuples waits if no new tuples are currently available, and wrappers are not done processing source contents. Once new records are available, they are pushed into plan execution by notifying all waiting requesters.

Each mashup source has a dedicated *synchronized buffer* satisfying these two requirements. Synchronized buffer owns a monitor (lock) to prevent concurrent reads and updates. Wrappers writing to the buffer, as well as mashup plan nodes, reading from the buffer, must obtain buffer's lock before accessing its data. If a read/write request is being served, all other requests are forced to wait until the request being served completes. As soon as the request completes, a notification message is issued to wake up all waiting requests to re-attempt accessing the buffer.

MashRank executor interleaves extraction with query processing such that none of the two tasks blocks the other (Challenge 2 in Section 1.1). Moreover, variance in source response times is tolerated by allowing asynchronous updates as soon as source responds with contents, as opposed to blocking until source responds. Hence, the execution is geared toward early-out of mashup results, if possible, while extraction and query processing are in progress.

4. UNCERTAIN RANK JOIN

Based on MashRank relational model, we show how to support rank join queries on uncertain scores (Challenge 3 in Section 1.1). We describe our scoring model (Section 4.1), problem definition (Section 4.2), and proposed solution (Sections 4.3 and 4.4).

4.1 Scoring Model

Without loss of generality, we model the score of tuple t_i as a random variable with possible values in the interval $[lo_i, up_i] \subseteq [0, 1]$. We adopt the convention that higher score values are preferred. Single-valued scores are represented as score intervals with equal bounds. The score random variable of tuple t_i has a PDF P_i encoding the likelihood of possible scores of t_i. We assume that the random variables representing the scores of base tuples are independent. The score densities P_i's can be computed by sampling the scores of tuples similar to t_i in their attribute values [23], or constructed from histories and value correlations as in sensor nets [10] and location tracking [5, 6]. We mean by "uncertain score" the interval-based score representation described above.

The authors in [16] assume a similar model, where *generating functions* are used to formulate and compute ranking queries efficiently on continuous score distributions. Our work mainly addresses consequences of assuming such model in the case of joins.

The scoring model leads to an intuitive way to rank tuples with uncertain scores, as formalized by the following definition:

DEFINITION 1. **[Score Dominance]** *We say that tuple t_i dominates another tuple t_j, denoted $(t_i \succ t_j)$, iff $lo_i \geq up_j$.* □

When $t_i \succ t_j$, we rank t_i on top of t_j. However, when $t_i \not\succ t_j$ and $t_j \not\succ t_i$, the relative order of t_i and t_j needs to be defined using additional means, as we propose in Section 4.2. Furthermore, for two different tuples t_i and t_j with equal single valued scores (i.e., $lo_i = up_i = lo_j = up_j$), we assume a tie-breaker $\tau(t_i, t_j)$ that gives a deterministic relative order. That is, $\tau(t_i, t_j)$ decides whether $t_i \succ t_j$ or $t_j \succ t_i$. We assume that the tie-breaker τ is transitive, and hence no cycles can arise when applying Definition 1 to tuples with equal single-valued scores. One example for such tie-breaker is ordering based on unique tuples IDs.

Our adoption of score dominance as a comparison criterion is supported by intuitive properties for meaningful comparison: It is straightforward to see that score dominance is non-reflexive (i.e., $t_i \not\succ t_i$), transitive (i.e., $((t_i \succ t_j), (t_j \succ t_k)) \Rightarrow (t_i \succ t_k)$), and asymmetric (i.e., $(t_i \succ t_j) \Rightarrow (t_j \not\succ t_i)$). It follows that a partial order holds on tuples with uncertain scores [22].

Uncertain scores induce a space of tuple orderings. Specifically, given a relation $R = \{t_1, \ldots, t_n\}$, let ω be an ordering of R tuples, where $\omega[t_i]$ is the rank of t_i in ω. We say that ω is a valid ordering of R iff $(\omega[t_i] < \omega[t_j]) \Rightarrow (t_i \succ t_j)$ or $(t_i \not\succ t_j \wedge t_j \not\succ t_i)$. The valid orderings are equivalent to possible linearizations (topological sorts) of a partial order. The number of possible orderings is exponentially large in general. The orderings space is generated by a probabilistic process that draws, for each tuple t_i, a score $s_i \in [lo_i, up_i]$ based on the density P_i. Ranking the drawn scores gives an ordering whose probability is the joint probability of drawn scores. That is, the probability of an ordering $\omega = \langle t_1, t_2, \ldots t_n \rangle$ is computed as follows:

$$\Pr(\omega) = \int_{lo_1}^{up_1} \int_{lo_2}^{x_1} \ldots \int_{lo_n}^{x_{n-1}} \mathcal{P}(x_1, x_2, \ldots, x_n)\, dx_n \ldots dx_1 \quad (1)$$

where $\mathcal{P}(.)$ is the PDF of the joint distribution of $P_1, \ldots, P_n$. When the score random variables are independent, we have $\mathcal{P}(x_1, \ldots, x_n) = \Pi_{i=1}^{n} P_i(x_i)$.

Figure 4 shows a relation R with uniform uncertain scores. The relation R has 7 possible orderings $\{\omega_1, \ldots, \omega_7\}$. The probabilities of ω_i's are computed by evaluating Equ 1 using Monte-Carlo integration (cf. Appendix A), while assuming independence of score densities. We discuss the computation of the measures $\mathrm{E}[t]$ and $\mathrm{ER}[t]$, shown in Figure 4, in Section 4.2.

4.2 Problem Definition

We assume a monotone user-defined scoring function $\mathcal{F}$ to be the source of the scores of join results (i.e., $\mathcal{F}(x_1, \ldots, x_n) \geq \mathcal{F}(\acute{x}_1, \ldots, \acute{x}_n)$ whenever $x_i \geq \acute{x}_i$ for every i). Typical scoring functions, such as summation, multiplication, min, max, and average, are monotone functions.

DEFINITION 2. **[Uncertain Rank Join (URANKJOIN)]** *Let $\mathcal{R}$ be a set of relations $\{R_1, \ldots, R_m\}$, $\mathcal{F} : R_1 \bowtie \ldots \bowtie R_m \rightarrow \mathcal{I}$ be a monotone scoring function, where $\mathcal{I}$ is the domain of all possible*

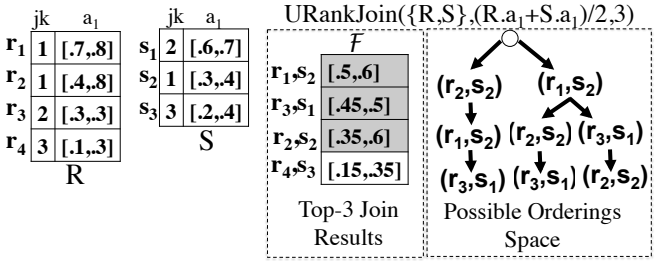


Figure 5: URANKJOIN **example**

sub-intervals of [0,1], and k be an integer $\leq |R_1 \bowtie \ldots \bowtie R_m|$. The query URANKJOIN$(\mathcal{R}, \mathcal{F}, k)$ *computes a total order ω^* under some probabilistic ranking semantics (described below) of tuples in the set $\mathcal{J}_k \subseteq R_1 \bowtie \ldots \bowtie R_m$, where $|\mathcal{J}_k| \geq k$, and*
$\forall t_i \in \mathcal{J}_k$: $|\{t_j \in R_1 \bowtie \ldots \bowtie R_m : t_j \succ t_i\}| < k$, and
$\forall t_j \notin \mathcal{J}_k$: $|\{t_i \in \mathcal{J}_k : t_i \succ t_j\}| \geq k$. □

That is, URANKJOIN returns an ordering of tuples that have less than k other dominating tuples. To illustrate, consider Figure 5. URANKJOIN$(\{R, S\}, (R.a_1 + S.a_1)/2, 3)$, where the join condition is equality of attribute 'jk', returns a total order of join tuples in $\mathcal{J}_3 = \{(r_1, s_2), (r_3, s_1), (r_2, s_2)\}$, since all join tuples in $\mathcal{J}_3$ are dominated by less than 3 join tuples, and all join tuples not in $\mathcal{J}_3$ (only (r_4, s_3) in this example) are dominated by at least 3 tuples. Based on the monotonicity of $\mathcal{F}$, the lo and up scores of join tuples are given by applying $\mathcal{F}$ to the lo and up scores of the corresponding base tuples. For example, the score of (r_1, s_2) is given by $[\mathcal{F}(.7, .3), \mathcal{F}(.8, .4)] = [\frac{.7+.3}{2}, \frac{.8+.4}{2}] = [.5, .6]$.

Computing $\mathcal{J}_k$ does not require knowledge of P_i's of base or join tuples, since $\mathcal{J}_k$ is based on score dominance only. However, computing ω^* requires knowledge of P_i's.

We view computing ω^* as a sophisticated tie-breaking rule that maps tuples with overlapping score distributions to a total order. Total order is a widely accepted means for presenting a ranking (e.g., on the Web, results are usually totally ordered based on relevance to user's query). We thus assume total order as an easier to grasp presentation of results. Nevertheless, computing a total order is an added feature of the techniques we propose, since we can stop at computing $\mathcal{J}_k$ if results incomparability is not a concern.

Probabilistic Ranking Semantics. Let Ω be the set of all possible orderings of tuples in $\mathcal{J}_k$, and $\omega[t]$ be the rank of t in an ordering $\omega \in \Omega$ (Section 4.1). An intuitive requirement in the total order ω^* is that it complies with score dominance (i.e., $(t_i \succ t_j) \Rightarrow (\omega^*[t_i] < \omega^*[t_j])$). Multiple semantics can be adopted to satisfy this requirement, as formalized in the following:

(1) Expected Scores. Let $\mathrm{E}[t_i] = \int_{lo_i}^{up_i} x \cdot P_i(x)dx$. Then, $\omega^*[t_i] = 1 + |\{t_j : \mathrm{E}[t_j] > \mathrm{E}[t_i]\}|$, while resolving ties deterministically. For example in Figure 4, based on expected scores, $\omega^* = \langle t_5, t_1, t_2, t_3, t_4, t_6\rangle$, assuming that the tie between t_1 and t_2 is resolved in favor of t_1.

(2) Expected Ranks. Let $\mathrm{ER}[t_i] = \sum_{\omega \in \Omega} \omega[t_i] \cdot \Pr(\omega)$. Then, $\omega^*[t_i] = 1 + |\{t_j : \mathrm{ER}[t_j] < \mathrm{ER}[t_i]\}|$, while resolving ties deterministically. In Figure 4, based on expected ranks, $\omega^* = \langle t_5, t_2, t_1, t_3, t_4, t_6\rangle$. The same definition is used in [7] with the addition that tuples can be excluded from some orderings due to their membership uncertainty.

(3) Most Probable Ordering. In [22] ω^* is defined as $argmax_{\omega \in \Omega} \Pr(\omega)$, where $\Pr(\omega)$ is computed using Equ 1. For example in Figure 4, ω^* is the ordering $\omega_1 = \langle t_5, t_1, t_2, t_3, t_4, t_6\rangle$.

Contrasting the properties of different semantics of ω^*, and studying their potential applications have been addressed in recent works [7, 22, 24, 26]. Our focus in this paper, however, is building an infrastructure that incrementally computes both $\mathcal{J}_k$ and ω^* under multiple semantics in the context of URANKJOIN queries.

4.3 Computing the Top-k Join Results

Tuples dominated by less than k tuples can be computed based on Theorem 1:

THEOREM 1. *Let $R = \{t_1, \ldots, t_n\}$ be a set of tuples with uncertain scores. Let $\mathcal{T}$ be the set of tuples in R, where each tuple in $\mathcal{T}$ is dominated by less than k tuples. Let $t_{(k)} \in R$ be the tuple with the k^{th} largest score lower bound. Then, $\forall t_i \in R$ we have: (1) $(up_i > lo_{(k)}) \Rightarrow t_i \in \mathcal{T}$, and (2) $[(up_{(k)} = lo_{(k)} = up_i = lo_i) \wedge (\tau(t_i, t_{(k)}) = t_i)] \Rightarrow t_i \in \mathcal{T}$.* □

That is, tuples in $\mathcal{J}_k$ are obtained by pruning all join result dominated by $t_{(k)}$; the join results with the k^{th} largest score lower bound. We include the proof of Theorem 1 in Appendix B.

We describe how to compute and sort join results incrementally (as needed) using a rank join algorithm to early prune all dominated tuples.

A common interface to most rank join algorithms, is to assume input relations sorted on per-relation scores, while output (join) relation is generated incrementally in join scores order. We show how to use a generic rank join algorithm RJ, complying with the previous interface, as a building block to compute $\mathcal{J}_k$ incrementally.

Our algorithm COMPUTE-$\mathcal{J}_k$ assumes two sorted inputs (e.g., indexes) L^i_{lo} and L^i_{up}, for each input relation R_i, giving relation tuples ordered on lo and up scores, respectively. By processing the lo and up inputs simultaneously, COMPUTE-$\mathcal{J}_k$ incrementally computes $\mathcal{J}_k$. This is done by using two instances of RJ, denoted RJ_{lo} and RJ_{up}, where RJ_{lo} rank-joins tuples on their overall lo scores to find exactly k join results, while RJ_{up} rank-joins tuples on their overall up scores to find all join results with up scores above the k^{th} largest score reported by RJ_{lo} (cf. Theorem 1). The execution of RJ_{lo} and RJ_{up} is interleaved, such that, at any point during execution, RJ_{up} reports all tuples whose up scores are above the last lo score reported by RJ_{lo}. Tuples in $\mathcal{J}_k$ are reported in up scores order to allow for incremental ranking (cf. Section 4.4). More details on COMPUTE-$\mathcal{J}_k$ are included in Appendix B.

Pipelined Operator. MashRank Planner generates pipelined URANKJOIN query plans by wrapping COMPUTE-$\mathcal{J}_k$ into a query operator. For clarity of presentation, we focus on 2-way joins plans. However, our techniques can also handle multi-way joins. A pipelined operator implementation of COMPUTE-$\mathcal{J}_k$ requires making the algorithm independent of k. The knowledge of k is only available to the query plan root that drives plan execution, while the operator only responds to incoming requests of join results ordered on either lo or up scores.

A URANKJOIN plan is rooted by ULIMIT, a new operator we propose to drive URANKJOIN plan execution. The operator takes two inputs I_{lo} and I_{up} representing two streams of query output tuples ordered on their lo scores and up scores, respectively. One GETNEXT implementation is to consume k tuples from I_{lo} and to report tuples in I_{up} with scores above the k^{th} score in I_{lo}. An alternative GETNEXT implementation is to interleave drawing tuples from I_{lo} and I_{up}, similar to Algorithm COMPUTE-$\mathcal{J}_k$. We give the details of ULIMIT in Appendix B.

A URANKJOIN operator is a logical operator that accepts two inputs each has two sorted access paths, corresponding to the lo and up score orders of the two input relations. The operator produces two output tuple streams corresponding to sorted join results based on lo and up scores.

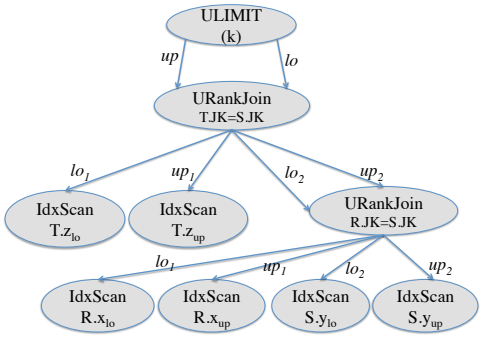


Figure 6: A logical URANKJOIN query plan

Figure 6 gives an example logical URANKJOIN query plan. The shown plan rank-joins three relations R, S, and T with uncertain scores x, y, and z, respectively. The bottom URANKJOIN operator uses indexes on the lo and up scores in Relations R and S as its input access paths, while the top URANKJOIN operator uses indexes on Relation T and the output of the bottom URANKJOIN operator as its input access paths. The ULIMIT operator consumes both lo and up inputs from the top URANKJOIN operator.

URANKJOIN operator can have different physical implementation. One implementation is to use two regular rank join operators wrapped within a physical operator with 4 inputs (the lo and up orders of the two input relations) and 2 outputs (the lo and up orders of the join results). This implementation requires, however, making other query operators aware of the URANKJOIN operator input/output interface.

An alternative implementation is to use two separate rank join operators, which allows building URANKJOIN plan as two parallel plans that can be optimized independently based on available data access paths. We use this implementation in MashRank prototype. The algebra proposed in [15] can be used in these settings to exploit properties like associativity and commutativity of rank join operators while searching for the query plan with least estimated cost. The logical design of URANKJOIN operator does not restrict the physical rank join algorithm. Hence, an arbitrary rank join algorithm can be plugged in physical URANKJOIN plans. Such flexibility raises interesting query optimization and cost modeling issues, which are part of our future work.

4.4 Ranking the Top-k Join Results

A major challenge for ordering tuples with uncertain scores is managing the exponentially large space of possible orderings (Section 4.1). We tackle this challenge using a sampling-based infrastructure for computing ω^* under multiple semantics based on two novel techniques: *Join-aware Sampling* and *Incremental Ranking*.

Join-aware Sampling. Join induces correlations among join results. For example in Figure 5, (r_1, s_2) and (r_2, s_2) are correlated, since they originate from one tuple $s_2 \in S$ and different tuples in R. Such correlation means that the joint score density of (r_1, s_2) and (r_2, s_2) (which produces the probability of any ordering involving (r_1, s_2) and (r_2, s_2)) is not given by multiplying the marginal score densities of (r_1, s_2) and (r_2, s_2). That is, the score random variables of (r_1, s_2) and (r_2, s_2) are dependent.

We handle score correlations by associating join results with lineage representing the keys of their origin base tuples. The main idea is to use the space with independent score random variables (i.e., base tuples) as a generator of the space with correlated score random variables (i.e., join results). Hence, the probability of an ordering of (possibly correlated) join results is computed using independent samples drawn from the space of base scores.

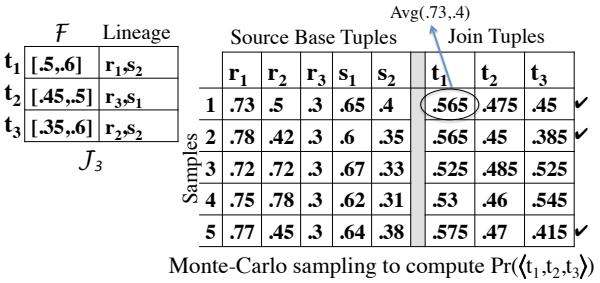


Figure 7: Handling score correlations in Monte-Carlo sampling

Figure 7 illustrates our approach. The set $\mathcal{J}_3$ is produced by the URANKJOIN query in Figure 5. Each join result in $\mathcal{J}_3$ is associated with the keys of its origin base tuples. To compute $\Pr(\langle t_1, t_2, t_3\rangle)$, we independently sample a score value in each origin base tuple of $\mathcal{J}_3$ (i.e., a score value in each of $\{r_1, r_2, r_3, s_1, s_2\}$). We simply call such vector of base tuples' score samples a *base sample*. Since base tuple scores are independent, the probability of each base sample is the product of the probabilities of its constituent score values[†]. Applying $\mathcal{F}$ (the average) to score values in a base sample gives a score sample for each join results in $\mathcal{J}_3$. We mark in Figure 7 the base samples that correspond to the ordering $\langle t_1, t_2, t_3\rangle$. Such base samples are called *hits*.

We use Monte-Carlo integration to estimate the probability of an ordering of join results based on the proportion of its corresponding hits with respect to the number of samples. We describe Monte-Carlo integration in more detail in Appendix A, and elaborate on the details of our probability computation method in Algorithm 3 in Appendix B.

We show how to use the previous infrastructure to compute the total order ω^* of tuples in $\mathcal{J}_k$ under *expected ranks* semantics. We discuss other semantics in Appendix C.

The expected rank of a tuple t_i ($ER(t_i)$) is computed as $\sum_{\omega\in\Omega} \Pr(\omega) \cdot \omega[t_i]$ (cf. Section 4.2). Computing $ER(t_i)$ can be done efficiently in our settings by rewriting $ER(t_i)$ as $\sum_{r=1}^{|\mathcal{J}_k|} r \cdot \Pr(t_i, r)$, where $\Pr(t_i, r)$ is the probability of t_i to be at rank r in the possible orderings of $\mathcal{J}_k$. The correctness of the previous rewrite follows from the fact that $\Pr(t_i, r) = \sum_{\omega_{(t_i,r)}\in\Omega} \Pr(\omega_{(t_i,r)})$, where $\omega_{(t_i,r)}$ has t_i at rank r.

We compute $\Pr(t_i, r)$ by considering as a *hit* each sample of base scores that results in the join tuple t_i being at rank r. For example in Figure 7, to compute $\Pr(t_3, 1)$, we consider sample 3 and sample 4 as *hits*.

Incremental Ranking. The size of $\mathcal{J}_k$ can be much larger than k due to score uncertainty. In many Web application scenarios, users only inspect a small prefix of the ranked answers list (e.g., inspecting only a few top hits returned by a search engine). Computing a full ranking of all answers in advance may not thus be always required. We make use of the incremental computation of $\mathcal{J}_k$ (cf. Section 4.3) to incrementally compute an approximation of ω^*.

The main idea is computing bounds on $\Pr(t_i, r)$ for each join result t_i produced by a URANKJOIN plan. The bounds of $\Pr(t_i, r)$ are used to approximate a prefix of ω^*, and are progressively tightened as more tuples are produced by the URANKJOIN plan.

Figure 8 shows a URANKJOIN plan that produces tuples in $\mathcal{J}_k$ ordered on their up scores. The last produced tuple at this step is t_3. Assume that we need to compute $\Pr(t_2, r)$. We identify two extreme cases:

- Case (i): the scores of all non-retrieved tuples are determin-

[†]For a continuous P_i, a sample score value v_i is an interval in $[lo_i, up_i]$ with a small predefined width ϵ, and $P_i(v_i)$ is the integration of P_i over v_i.

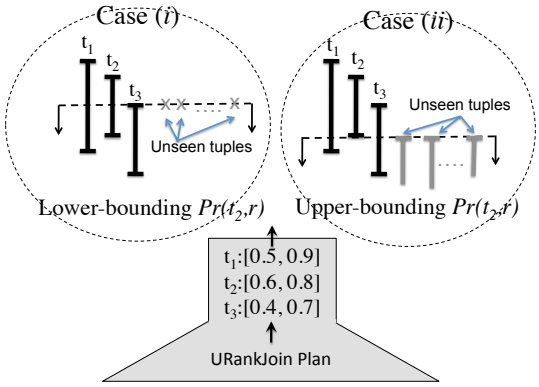


Figure 8: Bounding $\Pr(t_2, i)$

istic values (shown as '$\times$' symbols in Figure 8) located at the largest possible unseen score (i.e., up_3).

- Case (ii): the up scores of all non-retrieved tuples are below lo_2 (shown as shaded intervals in Figure 8).

Each case gives a possible configuration of unseen tuples in $\mathcal{J}_k$. By applying Monte-Carlo sampling to each configuration, we obtain a bound on $\Pr(t_2, r)$. Specifically, Case (i) gives a lower bound on $\Pr(t_2, r)$, denoted $\underline{\Pr}(t_2, r)$, since the scores of all unseen tuples are maximized, while Case (ii) gives an upper bound on $\Pr(t_2, r)$, denoted $\overline{\Pr}(t_2, r)$, since the scores of all unseen tuples are minimized. By seeing more tuples in $\mathcal{J}_k$, computed bounds are tightened (i.e., $\underline{\Pr}(t_2, r)$ increases and $\overline{\Pr}(t_2, r)$ decreases) since the maximum score of an unseen tuple decreases. When the maximum score of an unseen tuple is below lo_2, both bounds coincide at $\Pr(t_2, r)$. Note that this bounding method is valid only if tuples in $\mathcal{J}_k$ are retrieved in up score order.

The bounds of $\Pr(t_i, r)$ can be used to compute rankings under multiple probabilistic ranking semantics. For example, in Global top-k [26] (rank by tuple's probability to appear within the top k ranks), we bound $\Pr_k(t_i)$ as $\underline{\Pr_k}(t_i) = \sum_{r=1}^{k} \underline{\Pr}(t_i, r)$, while $\overline{\Pr_k}(t_i) = \min(1, \sum_{r=1}^{k} \overline{\Pr}(t_i, r))$. The Global top-$k$ ranking of retrieved tuples from $\mathcal{J}_k$ can thus be approximated as follows. We set $\omega^*[t_i] < \omega^*[t_j]$ if $\underline{\Pr_k}(t_i) > \overline{\Pr_k}(t_j) - \epsilon$, where $\epsilon \in [0, 1)$ is a given acceptable error in tuples relative order. The underlying URANKJOIN plan is incrementally requested for new join results until the computed ω^* prefix satisfies the previous error constraint.

5. EXPERIMENTS

MashRank is a Web-accessible system (demonstrated in [25]), with client-side query processing, and server-side data retrieval and extraction. All experiments are conducted on a 2.2GHz client machine with 2GB RAM, and 80GB hard disk. While many mashup examples are implementable using MashRank, we use 4 examples to show effectiveness and scalability. Figure 9 gives details of involved Web sources, while Figure 10 shows (in SQL-like syntax) the mashup examples we build. Join conditions and scoring functions are selected based on source schemas. We assume uniform score distributions for all uncertain scores (in general, uncertain join scores can be non-uniform even if base scores are uniform). While MashRank handles other source types (e.g., relational and XML sources), our experiments focus on unstructured HTML to expose the full spectrum of problem challenges.

Our performance metrics are (1) *latency*: time before returning first result, (2) *overall time*, (3) *e-cost*: average extraction time per page, and (4) *quality*: prec./recall of information extraction. We control values of three parameters: (1) k: number of tuples dominating any produced result is $< k$ (default is 1), (2) *npages*: maximum number of pages per source, (default is 10), (3) *nsamples*: number of Monte-Carlo samples, (default is 10,000). When changing one parameter value, we keep other parameters at their defaults.

5.1 Scalability with respect to Source Size

We evaluate MashRank performance as *npages* increases. In general, processing time increases sub-linearly with *npages*. Figure 11 shows that *overall time* has increased by an average of 5.5 times, as *npages* increased by 10 times. Since MashRank computes mashups out of records extracted on-the-fly, most of processing time is consumed in grabbing data from online sources. Figure 12 shows the average *get time* consumed in grabbing a source page. Since grabbing data from multiple pages is parallelized in MashRank (cf. Section 3), as *npages* increased by 10 times, *get time* increased by an average of 4.5 times, due to the added overhead of grabber threads synchronization. We also measured *e-cost*, which had a value below 0.5 seconds in all sources.

5.2 Push-based Mashup Execution

We compare push-based execution (cf. Section 3) to *pull-based* execution, where extraction from all sources needs to finish before query processing starts. We removed ranking requirements of mashups in this experiment to maximize the chances of results pipelining. Figure 13 compares the *latency* of the two models for different mashups, while Figure 14 compares the *overall time*. Push-based execution improves *latency* over pull-based with an average of 69%, and improves *overall time* with an average of 41%. The savings are due to the ability of conducting query processing on extracted records, while other extraction operations are undergoing, as opposed to blocking until all extraction completes, or waiting on slow sources to respond.

5.3 Sort Offloading in Rank-aware Processing

We evaluate the performance impact of sort offloading (cf. Appendix E). In M1, we exploit sorted access of the Vianet source to avoid sorting its records, and pipeline them into plan execution. We compare the generated plan to the conventional materialize-sort plan (i.e., compute all mashup results then sort) that does not exploit pre-sorted records. Figure 15 shows that sort offloading and rank-aware processing improved *latency* by an average of 66% over different *npages* values, while Figure 16 shows that the average *latency* improvement is 56% over different k values.

5.4 Monte-Carlo Sampling

Figure 18 illustrates convergence of ω^* under expected ranks semantics (cf. Section 4.4). We use normalized Kendall tau distance as our convergence measure. Kendall tau distance between two orderings is the number of pairs of items with disagreeing relative orders in the two orderings. We measure Kendall tau distance between each two orderings produced using two consecutive sample sizes. As *nsamples* increases, the distance decreases indicating that ω^* approaches stability. Figure 17 shows the time consumed in generating MC samples of join tuples. The sampling algorithm shows linear increase in time with respect to *nsamples*.

6. CONCLUSION

To the best of our knowledge, MashRank is the first system to address integrating information extraction with joining and ranking under uncertainty in the context of Web mashups. Our proposal initiates a study of the integration of information extraction, relational

Source	Base URL	Schema	Description
Vianet	www.vianet.travel/search/list	Hotel, City, Price	Hotel booking
TvTrip	www.tvtrip.com	Hotel, City, Rating	Hotel reviews
Menus	www.menus.co.nz/wining-dining	Restaurant, City, Rating	Restaurant reviews
Epinion	www.epinions.com/ Digital_Cameras	Brand, MegaPixels, Price	Camera offers
Flickr	www.flickr.com/cameras	Brand, nUsers, Rank	Camera usage info
Pubs	www.cs.uwaterloo.ca/~ilyas/ publistC.html	PaperTitle	A publications page
GScholar	scholar.google.com/scholar? q=author:ihab-ilyas	Paper, nCitations	Citations count
Apartments	www.apartments.com	Price, Zip, Info, Tel	Apartment search
Restaurants	www.restaurantica.com	Name, Cusine, Tel, Zip	Restaurant search

Figure 9: Web sources used in experiments (boxes indicate uncertain attributes)

M1

```
SELECT *
FROM Vianet v, TvTrip t,
     Menus m
WHERE v.Hotel ≈ t.Hotel
     AND v.City=m.City
ORDER BY 500-v.Price+ 100*
(t.Rating+m.Rating)
LIMIT k
```

M2

```
SELECT *
FROM Epinion e, Flickr f
WHERE e.Brand contains
f.Brand
ORDER BY e.Price+
        (100-f.Rank)
LIMIT k
```

M3

```
SELECT *
FROM Pubs p, GScholar g
WHERE p.PaperTitle ≈ g.Paper
ORDER BY nCitations
LIMIT k
```

M4

```
SELECT *
FROM Apartments a,
Restaurants r
WHERE a.Zip = r.Zip
ORDER BY a.Price
LIMIT k
```

Figure 10: Mashup examples used in experiments

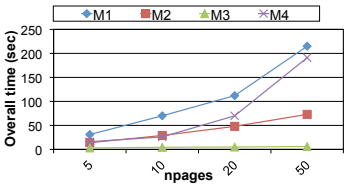


Figure 11: Overall execution time

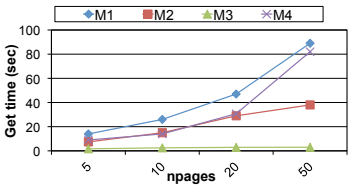


Figure 12: Avg source grabbing time

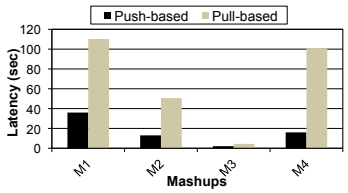


Figure 13: Execution models latency

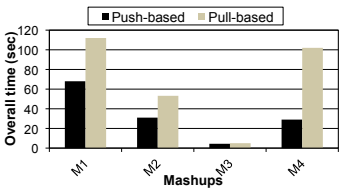


Figure 14: Execution models time

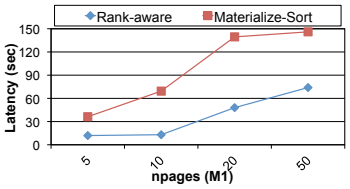


Figure 15: Sort offloading

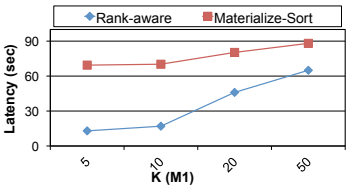


Figure 16: Sort offloading with k

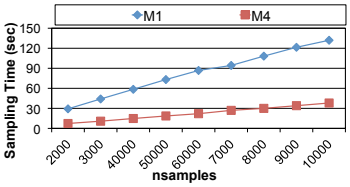


Figure 17: MC time

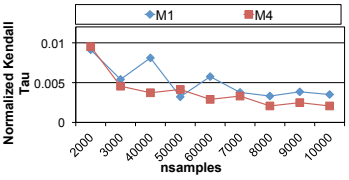


Figure 18: MC Convergence

optimizations (e.g., rank join), and the concepts of probabilistic databases towards building a relational+probabilistic query engine. We design an architecture that integrates information extraction with query processing within a push-based execution model. We give an implementation of probability and rank-aware join operators, and an infrastructure for uncertain rank-join under multiple probabilistic ranking semantics using Monte-Carlo simulation.

7. REFERENCES

[1] Google mashup editor: http://code.google.com/gme/.

[2] Microsoft Popfly: http://www.popfly.com/.

[3] Yahoo! Pipes: http://pipes.yahoo.com/.

[4] T. Anton. Xpath-wrapper induction by generalizing tree traversal patterns. *LWA 2005 - Workshopwoche der GI-Fachgruppen/Arbeitskreise*, 2005.

[5] S. Arumugam and C. Jermaine. Closest-point-of-approach join for moving object histories. In *ICDE*, 2006.

[6] R. Cheng, S. Prabhakar, and D. V. Kalashnikov. Querying imprecise data in moving object environments. In *ICDE*, 2003.

[7] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *ICDE*, 2009.

[8] V. Crescenzi, G. Mecca, and P. Merialdo. RoadRunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[9] N. N. Dalvi, P. Bohannon, and F. Sha. Robust web extraction: an approach based on a probabilistic tree-edit model. In *SIGMOD*, 2009.

[10] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *VLDB J.*, 14(4), 2005.

[11] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *SIGMOD*, 2008.

[12] I. F. Ilyas, W. G. Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3), 2004.

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[14] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *IJCAI (1)*, 1997.

[15] C. Li, K. C. Chang, I. F. Ilyas, and S. Song. RankSQL: query algebra and optimization for relational top-k queries. In *SIGMOD*, 2005.

[16] J. Li and A. Deshpande. Ranking continuous probabilistic datasets. In *VLDB*, 2010.

[17] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[18] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, 2008.

[19] P. Senellart, A. Mittal, D. Muschick, R. Gilleron, and M. Tommasi. Automatic wrapper induction from hidden-web sources with domain knowledge. In *WIDM*, 2008.

[20] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh. Damia: data mashups for intranet applications. In *SIGMOD*, 2008.

[21] D. E. Simmen, F. Reiss, Y. Li, and S. Thalamati. Enabling enterprise mashups over unstructured text feeds with infosphere mashuphub and systemt. In *SIGMOD*, 2009.

[22] M. A. Soliman and I. F. Ilyas. Ranking with uncertain scores. In *ICDE*, 2009.

[23] M. A. Soliman, I. F. Ilyas, and S. Ben-David. Supporting ranking queries on uncertain and incomplete data. *VLDB Journal*, 2010.

[24] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *ICDE*, 2007.

[25] M. A. Soliman, M. Saleeb, and I. F. Ilyas. Mashrank: Towards uncertainty-aware and rank-aware mashups. In *ICDE*, 2010.

[26] X. Zhang and J. Chomicki. On the semantics and evaluation of top-k queries in probabilistic databases. In *ICDE Workshops*, 2008.

APPENDIX

A. MONTE-CARLO INTEGRATION

We show how to compute the probability of a tuple ordering using Monte-Carlo integration. For an ordering $\omega = \langle t_1, t_2, \ldots, t_n \rangle$, let Γ be the n-dimensional hypercube that consists of all possible combinations of tuple scores in ω. That is, $\Gamma = ([lo_1, up_1] \times [lo_2, up_2] \times \cdots \times [lo_n, up_n])$. Let v be the volume of Γ, and s be the number of drawn samples from Γ, where each sample is an n-dimensional vector composed of a random score value drawn from each P_i. Let $x_1 \ldots x_m$ be the samples among such s samples whose score ordering gives the same tuple ordering of ω. Then,

$$\Pr(\omega) \approx \frac{m}{s} \cdot v \cdot \frac{\sum_{i=1}^{m} \Pr(x_i)}{m} \quad (2)$$

The main idea is using Γ, in which uniform sampling is easy, to estimate the volume to be integrated on in Equ 1. The expected value of the above approximation is the true integral value with an $O(\frac{1}{\sqrt{s}})$ approximation error. That is, the error is independent of the size of the orderings space, and depends only on the number of samples.

By assuming independence of score random variables, [22] computes sample probability ($\Pr(x_i)$ in Equ 2) by multiplying the marginal probabilities of samples' constituent scores. Moreover, if the tuple score densities P_i's are uniform, Equ 2 reduces to $\Pr(\omega) \approx \frac{m}{s}$. That is, $\Pr(\omega)$ is the proportion of samples that conform to the tuple ordering given by ω.

B. PROOFS AND ALGORITHMS PSEUDOCODE

Proof of Theorem 1. If $(up_i > lo_{(k)})$, then the number of score intervals with lo scores $\geq up_i$ is less than k. Then, $t_i \in \mathcal{T}$, and hence (1) follows.

If t_i and $t_{(k)}$ have equal single-valued scores and the tie breaker τ favors t_i to $t_{(k)}$, then t_i must appear before $t_{(k)}$ in the order of R on lo scores. Hence, there are less than k tuples with lo scores $\geq up_i$, and hence (2) follows. □

We next give the pseudocode and more details of the algorithms we present in this paper.

Algorithm 1 gives the details of our method to compute the set of top-k join results $\mathcal{J}_k$ (cf. Section 4.3).

Algorithm 2 gives the details of the ULIMIT operator that we use to drive the execution of ranked mashup plans with score uncertainty (cf. Section 4.3).

Algorithm 3 shows how to use Monte-Carlo integration method to compute $\Pr(\omega)$, where ω is an ordering of join results (cf. Section 4.4). The union of the lineage of join results in ω is first computed. Independent samples are drawn from the score distributions of base tuples included in the lineage. The drawn scores produce an ordering of join results. If such ordering agrees with ω, then the sample is a *hit*. The probabilities of hits corresponding to a join results ordering ω are averaged when using Monte-Carlo integration method to compute $\Pr(\omega)$.

Algorithm 1 Compute Top-k Join Results

COMPUTE-$\mathcal{J}_k(L^1_{lo}, L^1_{up}, \ldots, L^m_{lo}, L^m_{up}$:Ranked Inputs, k:Answer Size, $\mathcal{F}$:Scoring Function)

```
1  RJ_lo ← an instance of RJ (L^1_lo, ..., L^m_lo, k, F)
2  RJ_up ← an instance of RJ (L^1_up, ..., L^m_up, ∞, F)
3  active_lo ← TRUE ; active_up ← TRUE
4  T_up ← 1 {initialize score upper bound in RJ_up}
5  count ← 0 {number of results reported by RJ_lo}
6  while (active_lo OR active_up) do
7      if (active_lo) then
8          t ← get next result from RJ_lo
9          count ← count + 1
10         if (count = k) then active_lo ← FALSE
11         T_lo ← score of t
12     while ( T_up > T_lo) do
13         Report results available in RJ_up with scores > T_lo
14         T_up ← score upper bound in RJ_up
15     if (NOT active_lo AND T_up < T_lo) then
16         active_up ← FALSE
```

C. COMPUTING AN ORDERING OF JOIN RESULTS UNDER OTHER SEMANTICS

We extend our discussion in Section 4.4 by describing how to compute an ordering ω^* of join results under other probabilistic ranking semantics:

(1) The Most Probable Ordering. [22] proposed using Markov Chains Monte-Carlo (MCMC) methods to approximate the most probable ordering by drawing samples from the orderings space biased by probability. The main idea is that for a current sample (ordering) ω_i, and a newly proposed sample ω_{i+1}, we always accept ω_{i+1} if $\Pr(\omega_{i+1}) > \Pr(\omega_i)$, otherwise we accept ω_{i+1} with probability proportional to $\Pr(\omega_{i+1})/\Pr(\omega_i)$. The MCMC method provably converges to the target distribution of possible orderings, and hence it can be used as a generator of orderings biased by their probabilities. Algorithm 3 allows computing $\Pr(\omega)$, where ω is an ordering of join results, and hence applying the MCMC method to approximate the most probable ordering.

(2) Other Semantics. Computing $\Pr(t_i, r)$ allows computing ω^* under other probabilistic ranking semantics. For example, tuple's probability to appear at the top ranks only (Global Top-k [26]) is computed as $\Pr_k(t_i) = \sum_{r=1}^{k} \Pr(t_i, r)$. Similarly, pruning tuples whose probabilities to appear at the top ranks is below a given threshold (probabilistic Top-k threshold [11]) can be done by testing if $\Pr_k(t_i) < T$, for a given threshold T. A third example is finding the ordering with the minimum disagreements with other orderings in the space (Uncertain Rank Aggregation [23]), which can be done in polynomial time using $\Pr(t_i, r)$ values as shown in [23].

D. INFORMATION EXTRACTION

MashRank uses wrapper induction techniques to transform unstructured sources into relational (structured) sources. The details of the wrapper induction algorithm are orthogonal to mashup planning and processing in MashRank. We assume an interface to the wrapper inductor with three main functions: (1) *addExample*: adds a new training example (e.g., a text node representing the value of some attribute); (2) *learn*: processes the training examples using the induction algorithm to compute an extraction rule; and (3) *extract*: applies the learned extraction rule to a given page, and returns

Algorithm 2 ULIMIT Operator

OPEN(I_{lo}: lo input stream, I_{up}: up input stream, k: Answer Size)

```
1  I_lo.OPEN()
2  I_up.OPEN()
3  F_lo ← 1.0
4  count ← 0
```

GETNEXT()

```
1  while (count < k) do
2      t ← I_lo.GETNEXT()
3      count ← count + 1
4      if (count = k) then F_lo ← lo score of t
5  t ← I_up.GETNEXT()
6  if (up score of t > F_lo) then return t else return null
```

CLOSE()

```
1  I_lo.CLOSE()
2  I_up.CLOSE()
```

Algorithm 3 Compute Probability of Join Results Ordering

MC-PROBABILITY(ω: Join results ordering, s: Number of samples)

```
1  sources ← ⋃_{t∈ω} t.sources {compute lineage of ω}
2  hits ← 0 {no. of samples matching ordering given by ω}
3  sum ← 0 {summation of hits probabilities}
4  for i = 1 to s do
5      sample ← [ ] {sample of base tuples scores}
6      for each tuple t_i ∈ sources do
7          sample[t_i.key] ← random score value based on P_i
8      ώ ← ordering of join tuples based on base scores in sample
9      if (ώ agrees with the tuple ordering given by ω) then
10         hits ← hits + 1
11         sum ← sum + Π_{z∈sample}(Pr(z))
12 v ← volume of hypercube enclosing score combinations in ω
13 return (hits/s) · v · (sum/hits)
```

a set of extracted records.

The previous interface is generic, and applies to multiple wrapper induction proposals (e.g.,[4, 9, 14]). We elaborate on the implementation of the interface in our adaptation of [14]. We emphasize, however, that information extraction is a blackbox in MashRank, and hence other techniques can be integrated with MashRank to conduct more sophisticated extraction.

The inductor in [14] treats each HTML page as a sequence of characters, and learns extraction rules in the form of string patterns. The learned rule extracts attributes from the page source in a round-robin fashion, and binds them into records. This method can generate erroneous records when some attribute values are missing. Since missing values are common on the Web, we adapt this method by learning extraction rules on attribute level, and then bind extracted values into records based on their proximity in the HTML source. We describe our technique in the following.

For a schema $\langle a_1, \ldots, a_n \rangle$ of n attributes, the function *addExample* receives as input a triple (a_i, s, e), where a_i is a schema attribute, while s and e are the start and end character positions of one example value of a_i in the HTML source. In MashRank editor, this is enabled by allowing the user to highlight pieces of text inside the page as examples for each required attribute.

Source	Precision	Recall	F1
Vianet	1.0	1.0	1.0
TvTrip	1.0	0.92	0.95
Menus	0.97	0.96	0.97
Epinion	1.0	1.0	1.0
Flickr	0.92	1.0	0.96
Pubs	1.0	1.0	1.0
GScholar	1.0	0.94	0.97
Apartments	1.0	1.0	1.0
Restaurants	0.975	0.97	0.972

Figure 19: Information extraction precision/recall

The function *learn* computes an extraction rule for each attribute a_i in the form of a pair of strings (l_i, r_i). The rule is interpreted as follows: all values of attribute a_i appearing in the underlying page are enclosed between two strings l_i and r_i. For example, one possible extraction rule for hotel name could simply be ($" < b >"$, $" < /b >"$). The strings l_i and r_i are computed by scanning the characters appearing before and after all training examples, and appending these characters to l_i and r_i, respectively, as long as all examples agree on the scanned character. We stop when finding maximal patterns in the sense that by appending more characters to any of l_i and r_i, at least one training example is not matched.

The function *extract* applies the extraction rule of each attribute to extract a set of attribute values. We align extracted values to form records based on their proximity in the page. We process attributes in the order in which they appear in the HTML source (e.g., $name$ appears before $price$), and within each attribute, we process extracted values in the order of their appearance in the HTML source. We start by assigning each extracted value in the first attribute to a new record. For each subsequent attribute a_i, we assign attribute value v to the record that has an attribute a_j, with $j < i$, whose value is the closest value preceding v. If such record cannot be found, v is assigned to a new record with empty values in all attributes a_j for $j < i$, and value v in attribute a_i.

In our experimental study we evaluate extraction accuracy by counting as an error any extracted record with wrong information (e.g., missing values that should be non-missing, or wrongly linked values of different attributes). We manually computed a ground truth of all correct records to be extracted, and compared the output of our extraction technique to ground truth. Figure 19 shows extraction quality in precision, recall, and F1 measures, computed on a sample of 50 pages of each Web source used in our experiments (cf. Figure 9). We achieve perfect extraction in almost half of the sources, and very high accuracy on the rest. We note that the extraction method we adapt depends on regularity in HTML source, which may limit its applicability to some sources. However, as we discuss in Section 3, we treat information extraction as a blackbox, and hence other variants of extraction methods can also be plugged in MashRank framework.

E. MASHUP PLANNING

Query optimizers use statistics collected on queried relations, and query predicates, to prune query plans that are expected to perform poorly. In our settings, we usually have no prior knowledge about data sources, as they may be remote sources given by the user in an ad-hoc fashion. We thus resort to exploiting the configuration of mashup data flow to build a feasible mashup physical plan. Nev-

ertheless, building mashup planning on a cost model can be quite important in many other scenarios (e.g., mashing up sources that the system has prior knowledge on, asking for user input to characterize cost factors of mashed up sources, or sampling the sources to compute estimates on their cost factors). We leave cost modeling as an important future extension of this work.

Given a ranked mashup with a scoring dunction $\mathcal{F}$, MashRank Planner starts by labeling each node in the mashup data flow with its corresponding ranking attributes (attributes that appear in $\mathcal{F}$). The labeling starts with leaves (data sources), where each source is labeled with the ranking attributes it covers. Then, moving up in the data flow tree, the union of the ranking attributes of all children of a node p gives the ranking attributes of p.

After labeling is done, the Planner processes the labeled data flow starting from the root, mapping each node to one or more physical operators, and then recursing on nodes' children.

A source node is mapped to a *synchronized buffer* (cf. Section 3). An extractor node with empty ranking attributes is mapped to a *scan* operator. An extractor node with non-empty ranking attributes is mapped to *sort* operator, on top of a *scan* operator, so that all source tuples are sorted based on the scoring function (ranking attributes not belonging to the source assume the largest possible score). Since we assume monotone scoring functions, using such sort expression guarantees tuples flowing out of the source in the right order. A join node with empty ranking attributes is mapped to either a *nested-loops join* operator, or a *hash join* operator if the join condition is non-equality or equality, respectively. Similarly, a join node with non-empty ranking attributes is mapped to a *nested-loops rank join* operator, or a *hash rank join* operator[†] if the join condition is non-equality or equality, respectively. Finally, a filter node is mapped to a *filter* operator with the node's Boolean condition. We also implemented techniques to push down filtering operations to their relevant sources, as typically done in relational query optimizers.

When the scoring function includes one or more uncertain attributes, the Planner generates a URANKJOIN plan (cf. Section 4.3). The above procedure is followed to generate two identical rank join plans, where one plan rank-joins tuples on their lo scores, while the other plan rank-jons tuples on their up scores. A *ulimit* operator is used as the parent operator of the two plans, and a *probranker* operator (implementing our MC-based sampling methods) is added as the parent of *ulimit*.

We describe our plan generation algorithm using the rank join query given in Section 1.1:

```
 SELECT *
FROM vianet, tvtrip
WHERE vianet.HotelName ~ tvtrip.HotelName
ORDER BY 500-vianet.Price+ 100* tvtrip.Rating
LIMIT k
```

The scoring function includes two attributes *price*, and *rating*, where *price* is an uncertain attribute. The join condition is approximate equality of hotel names (implemented in MashRank as a thresholded edit distance similarity function). Figure 20 shows the data flow nodes after being labeled with ranking attributes. The generated physical plan is a nested-loops rank join plan (since join condition is non-equality). The Planner adds a *ulimit* operator to

[†]The Hash Rank Join (HRJN) algorithm [12] iteratively selects an input relation to read its next tuple. Each new tuple is hashed on its join attribute in a per-relation hash table to facilitate creating join results. The join results are created by finding, for each new tuple, the joinable tuples currently read from other relations. Join results are stored in a priority queue ordered on score. The scores of non-materialized join results are upper-bounded by assuming best-case joins, where tuples with the highest scores in all inputs, but one, join with the last retrieved tuple from the excluded input.

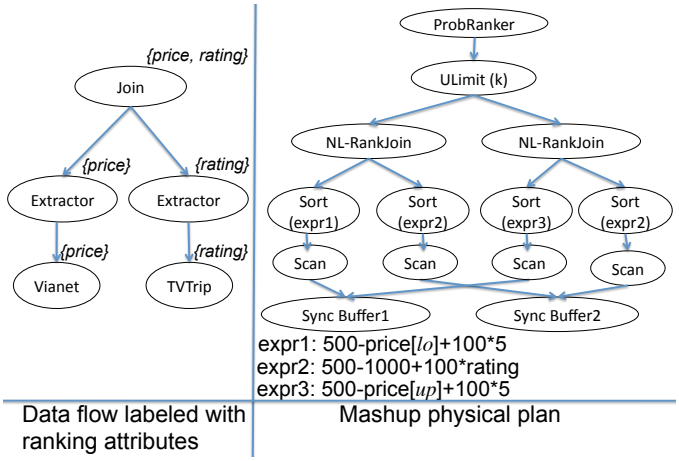


Figure 20: Generating mashup plan

drive the execution of the lo and up rank join plans, and a *probranker* operator to conduct probabilistic ranking. The sort expressions ($expr_1$, $expr_2$, and $expr_3$) are created by replacing ranking attributes not covered by the underlying source with their largest possible scores.

Offloading Sort to Web Sources. In rank-aware query processing, the existence of sorted access methods on ranking attributes is crucial for pipelining ranked results efficiently. Implementing such access methods as a sort operator per input (e.g., as in Figure 20) introduces a bottleneck in query execution due to the blocking nature of sort. When an index on a ranking attribute already exists, a rank-aware plan can benefit from such cheap sorted access method to pipeline ranked query results efficiently.

In our settings, we build mashups on arbitrary sources selected by the user, and hence we cannot generally assume the existence of indexes on these sources. However, many Web sources provide sorting capabilities to view query results ordered on some attribute. Such information is obtained from the user in the form of a special sorting parameter that can be appended to page requests (cf. Section 1). By offloading sort to source side, we allow rank-aware mashup plan to pipeline sorted results, as they are extracted from the sources.

For example, assume the following mashup query, where Vianet is declared by the user as a source that can produce records ordered on Price_up (the highest prices):

```
 SELECT *
FROM vianet, tvtrip
WHERE vianet.HotelName ~ tvtrip.HotelName
ORDER BY vianet.Price_up+ tvtrip.Rating
LIMIT k
```

Fiigure 21 shows the mashup data flow labeled with ranking attributes, and the corresponding physical plan generated by MashRank Planner. Note that the Planner did not add a sort on top of the scan of Vianet, since it leveraged the fact that records of Vianet are pre-sorted, and hence they can be directly pipelined into the NL-RankJoin operator.

We discuss how we modify our data grabbing module (cf. Section 3) to exploit the existence of sorted access methods. Our multithreaded architecture spawns a grabber thread per page to avoid blocking on slow sources. When each thread grabs data from one of the pages in a sorted retrieval, some of the pages can be ready

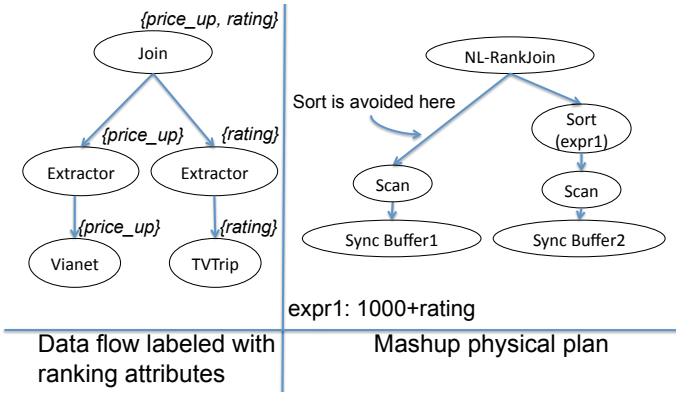


Figure 21: Generating mashup plan with offloaded sorting

for extraction before others, due to differences in source response time. We thus need to an maintain a page-level order to guarantee pipelining records into mashup execution in the right order. This is done by associating each thread with an order reflecting the position of the thread's page in the ordered retrieval of source pages. Tuple requests are answered while respecting such page-level order. That is, a tuple is not reported from page p unless all tuples in pages with orders preceding p have been already reported.