New Trends in Data Integration

Prof. Giovanni Simonini

Università degli Studi di Modena e Reggio Emilia

simonini@unimore.it

Outline

1. Join discovery at scale
    1. Single column: JOSIE
    2. Multi-columns: MATE

2. Language model for tabular data
    1. Tabular representation learning: TURL
    2. A unified multi-task model for matching: Unicorn

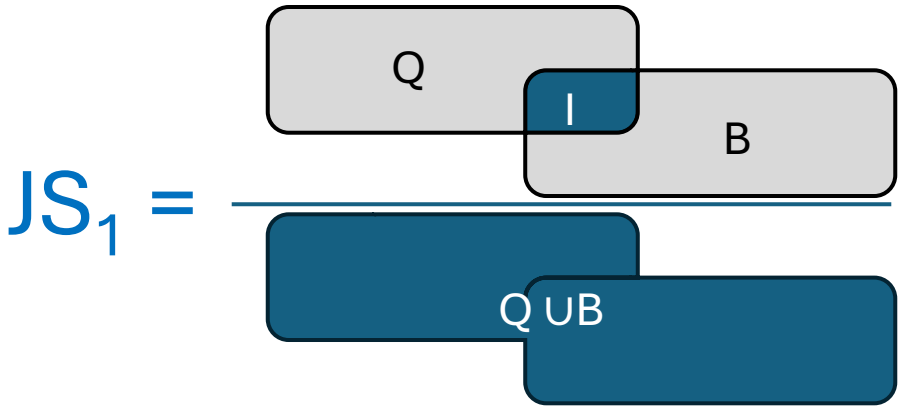
JOSIE

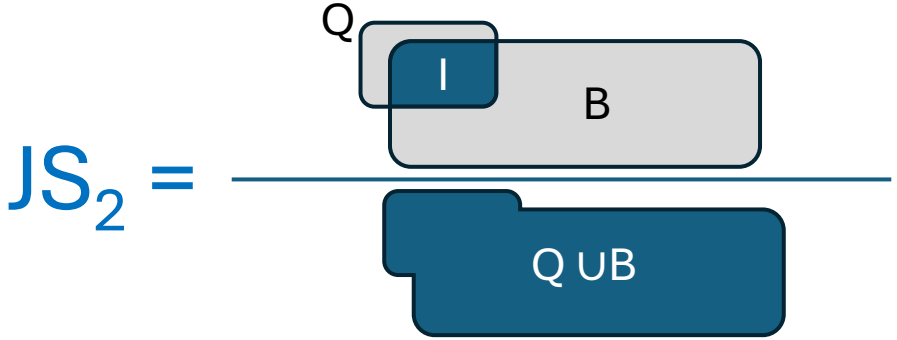
Overlap Set Similarity Search for Finding Joinable Tables in Data Lakes

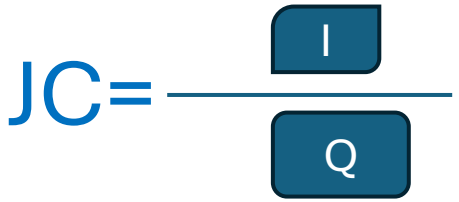
Zhu E, Deng D, Nargesian F, Miller RJ. Josie: Overlap set similarity search for finding joinable tables in data lakes. In Proceedings of the 2019 International Conference on Management of Data 2019 Jun 25 (pp. 847-864).

Overlap Set Similarity

- i.e., set intersection size
- Not biased against **sets of different size**
  ≠ Jaccard similarity, i.e., intersection over union
- Used in **previous work** - LSH Ensemble [1] - to detect **joinable tables** (table columns as sets of distinct values)
  - Approximate solution
    ≠ JOSIE (**exact**)
  - Threshold-based search
    ≠ JOSIE (**top-k**)
- Challenge:
  Data lakes have **large set sizes** and **huge dictionaries** (i.e., distinct values)







[1] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, Renée J. Miller: "LSH Ensemble: Internet-Scale Domain Search". Proceedings of the VLDB Endowment (PVLDB), 2016. https://doi.org/10.14778/2994509.2994534

LSH Ensamble (very semplified view)

1. Partition the corpus (e.g., the data lake tables) according to the size
   - The paper propose a partition strategy that optimize the final quelity

2. Compute MinHash index for each partition

3. At query time choose the proper “banding”
   - Size of Q and size of the Partition are known

JOSIE basic usage

An **exact top-k overlap set similarity search algorithm** that scales to large sets and dictionary sizes.

- All **columns** in all tables, as sets of distinct values, stored in a collection of **sets Ω**
- The user defines a **query table T_Q** and a **join column Q**
- JOSIE returns a sub-collection ω of the k columns in Ω with the largest set intersection with Q
  - $|Q \cap X| > 0, X \in \omega$
  - $\min\{|Q \cap X|, X \in \omega\} \geq |Q \cap Y|, Y \notin \omega, Y \in \Omega$ (top-k)

Preliminaries: Inverted Index and Dictionary

For a collection of sets X_1, X_2, … $\in \Omega$:

- Extract the tokens (i.e., attribute values) from all sets

- For each token, build a **posting list** of pointers to the sets that contain it
  - i.e., an inverted index
  - Implemented as a dictionary: token → frequency + pointer to its posting list
  - Very large, use disc (e.g., store in a DB)

Q = {Athens, Boston, Milan, Vancouver}

X_1 = {Athens, Milan, Vancouver}
X_2 = {Boston, Chicago}
X_3 = {Boston}
X_4 = {Boston, …, Milan, Naples}

$\Longrightarrow$

Athens : $\{X_1\}$
Boston : $\{X_2, X_3, X_4\}$
Milan : $\{X_1, X_4\}$
Vancouver : $\{X_1\}$

posting list

Preliminaries: Inverted Index and Dictionary

- **MergeList algorithm** (basic top-k search with inverted index):
  1. For every query token, use the dictionary to check if it exists in the inverted index
     - If exists, read its posting list and increment the counters for the sets in which it appears
  2. Sort the sets by their intersecting token counts and return the k sets with highest counts

  - E.g.:
    1. Athens is in the dictionary → counter[1] += 1
    2. Boston is in the dictionary → counter[2] += 1; counter[3] += 1; counter[4] += 1

    ...

    Top-2: X_1, X_4

Still too slow for very large datasets

Q = {Athens, Boston, Milan, Vancouver}

X_1 = {Athens, Milan, Vancouver}
X_2 = {Boston, Chicago}
X_3 = {Boston}
X_4 = {Boston, ..., Milan, Naples}

$\Longrightarrow$

Athens : $\{X_1\}$
Boston : $\{X_2, X_3, X_4\}$
Milan : $\{X_1, X_4\}$
Vancouver : $\{X_1\}$

posting list

Pruning candidates: **Prefix Filter (for top-k)**

ProbeSet algorithm (top-k search with prefix filter):

1. Use a **fixed-size heap** for keeping **current top-k candidate sets**
   - intersection size as a threshold
2. After reading a posting list:
   - check the **intersection size for the k-th candidate: $I_k = |Q \cap X_k|$**
3. If **$|Q| - I_k + 1$** posting lists already read:
   - **stop and return** the current top-k candidates
     - In fact, **only I_k - 1 posting** lists remain to read, so if a candidate does not appear yet in the heap, it cannot reach an intersection size equal to I_k
   - Do not read further posting lists (i.e., the first $|Q| - I_k + 1$ posting lists are the *prefix*)

X_1 = {Athens, Milan, Vancouver}
X_2 = {Boston, Chicago}
X_3 = {Boston}
X_4 = {Boston, …, Milan, Naples}

$\Longrightarrow$

$j_{X,0}$ |X|

Athens = {(X_1, 1, 3)}
Boston = {(X_2, 1, 2), (X_3, 1, 1), (X_4, 1, 100)}
Milan = {(X_1, 2, 3), (X_4, 99, 100)}
Vancouver = {(X_1, 3, 3)}

Pruning candidates: **Position Filter**

- Set a **global ordering** to the tokens and sort all sets accordingly
  - Heuristic: from the least frequent to the most frequent
- Add to the posting list:
  - the **position of the token in the sets that contain it** ($j_{X,0}$)
  - the **size of the sets |X|**
- For a new candidate X from the posting list of token x_i (i-th token in Q):
  - When a candidate is met for the first time:

    Compute the **upper bound** for its intersection size with Q:

$$|Q \cap X| \leq |Q \cap X|_{ub} = 1 + \min(|Q| - i, |X| - j_{X,0})$$

Tokens still to read from Q

Tokens still to read from X

  - If $|Q \cap X|_{ub} \leq |Q \cap X_k|$, skip X and ignore it when met again
  - Only need to read the **suffix** $X[j_{X,0} + 1:]$ to compute the exact intersection size

Preliminaries: Position Filter

Q = {Athens, Boston, Milan, Vancouver}

k = 2

X_1 = {Athens, Milan, Vancouver}
X_2 = {Boston, Chicago}
X_3 = {Boston}
X_4 = {Boston, …, Milan, Naples}

$\Longrightarrow$

$j_{X,0}$ $|X|$

Athens = {(X_1, 1, 3)}
Boston = {(X_2, 1, 2), (X_3, 1, 1), (X_4, 1, 100)}
Milan = {(X_1, 2, 3), (X_4, 99, 100)}
Vancouver = {(X_1, 3, 3)}

1. Read the posting list of *Athens*
   1. $|Q \cap X_1| = 3$ → push (X_1, 3) to the heap
2. Read the posting list of *Boston*
   1. $|Q \cap X_2| = 1$ → push (X_2, 1) to the heap
      → heap full: {(X_1, 3), (X_2, 1)}, $l_k = 1$, **activate the position filter**
   2. $|Q \cap X_3|_{ub} = 1 + \min(4 - 2, 1 - 1) = 1 \leq 1$ → skip
   3. $|Q \cap X_4|_{ub} = 1 + \min(4 - 2, 100 - 1) = 3 > 1$
      → compute $|Q \cap X_4| = 2$, push (X_4, 2) to the heap: {(X_1, 3), (X_4, 2)}, $l_k = 2$
3. Read the posting list of *Milan*, skip X_1 and X_4 (already seen)
4. Stop for the **prefix filter** ($|Q| - l_k + 1 = 4 - 2 + 1 = 3$)

$$|Q \cap X|_{ub} = 1 + \min(|Q| - i, |X| - j_{X,0})$$
$$|Q| - l_k + 1$$

Towards JOSIE

Q = {Athens, Boston, Milan, Vancouver}

k = 2

X_1 = {Athens, Milan, Vancouver}
X_2 = {Boston, Chicago}
X_3 = {Boston}
X_4 = {Boston, …, Milan, Naples}

$\Longrightarrow$

$j_{X,0}$ |X|

Athens = {(X_1, 1, 3)}
Boston = {(X_2, 1, 2), (X_3, 1, 1), (X_4, 1, 100)}
Milan = {(X_1, 2, 3), (X_4, 99, 100)}
Vancouver = {(X_1, 3, 3)}

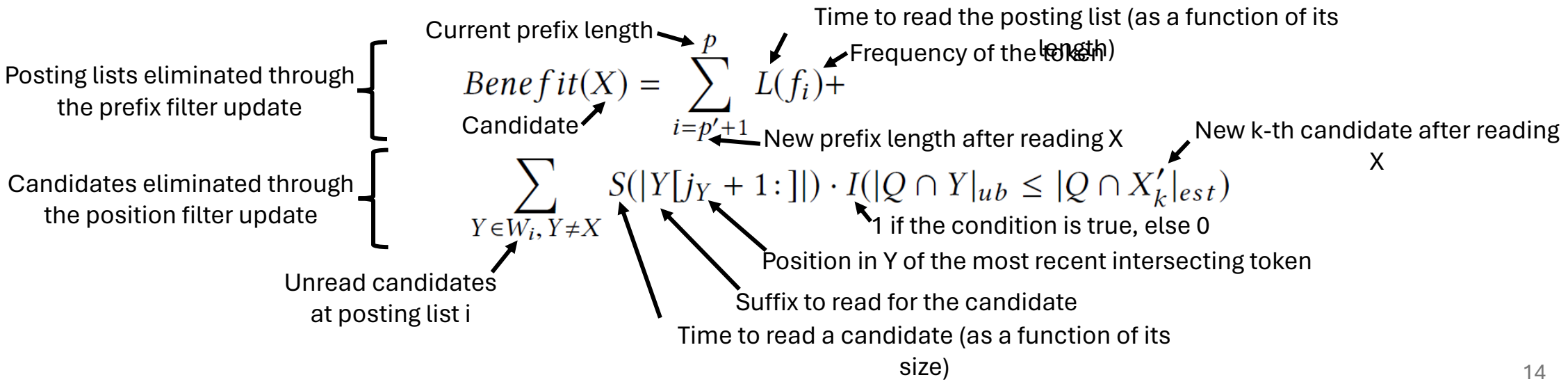
- After reading the posting list of *Boston*, read X_4 before X_2 and $X_3 \rightarrow l_k = 2$
  - Both X_2 and X_3 can be skipped through the **position filter** (upper bound ≤ 2)
  - Instead of checking the set X_2 and X_3, "follow" the token list of Q
- Prioritizing some candidates can **enhance the pruning power of the position filter** (i.e., read fewer candidates)
  - Trade-off between "following" the posting list and "following" the token list of Q
  - JOSIE propose a cost/benefit model for that

Reading Candidates

By **estimating the intersection size**, JOSIE quantifies the **benefits** (i.e., reduced costs) of reading candidates and posting lists

Potential benefit of reading candidates: increase the current k-th intersection size

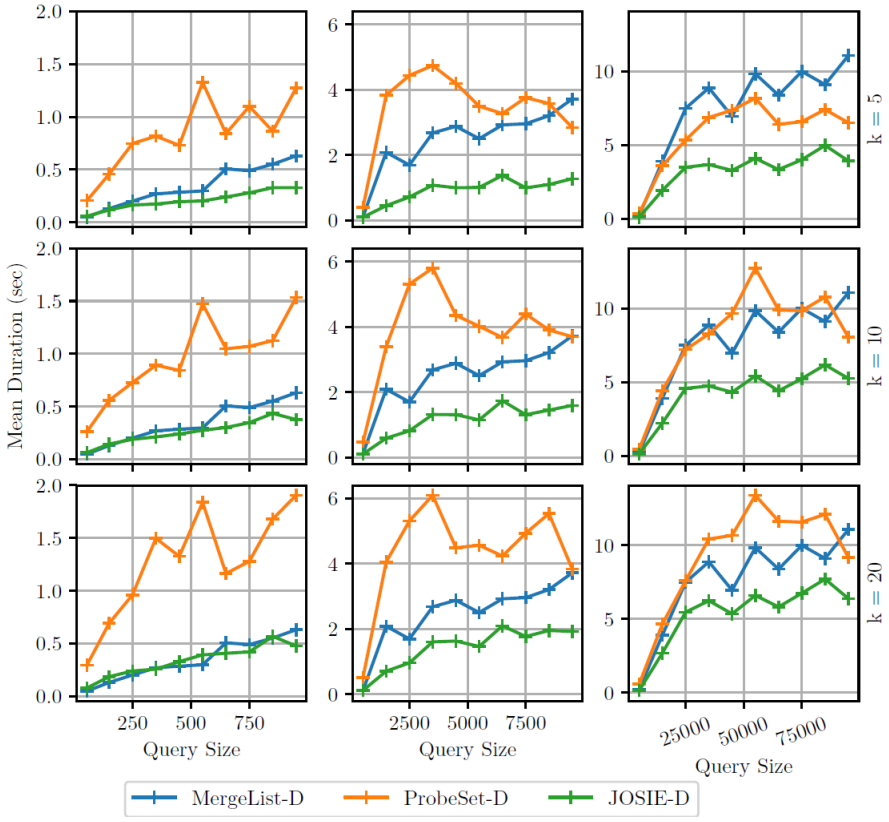
- Used by **prefix filter**: prune posting lists (and all unseen candidates there)
- Used by **position filter**: prune seen candidates without reading them



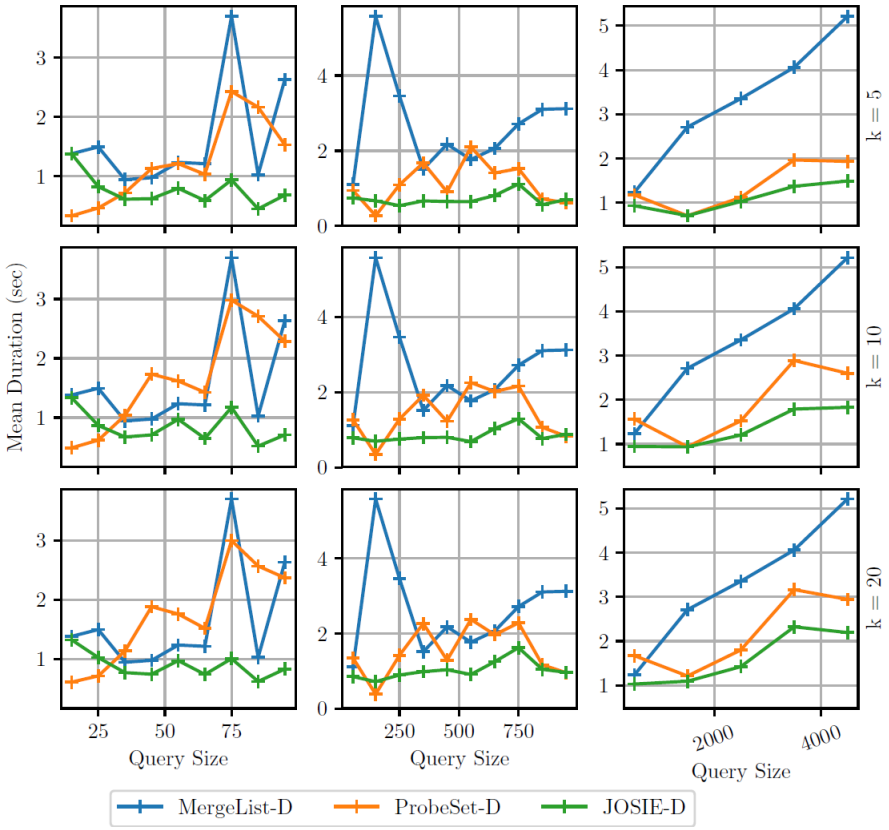
$$Benefit(X) = \sum_{i=p'+1}^{p} L(f_i) + \sum_{Y \in W_i, Y \neq X} S(|Y[j_Y + 1:]|) \cdot I(|Q \cap Y|_{ub} \leq |Q \cap X'_k|_{est})$$

JOSIE vs Baselines

	#Sets	MaxSize	AvgSize	#UniqTokens
OpenData	745,414	22,075,531	1,540	562,320,456
WebTable	163,510,917	17,030	10	184,644,583



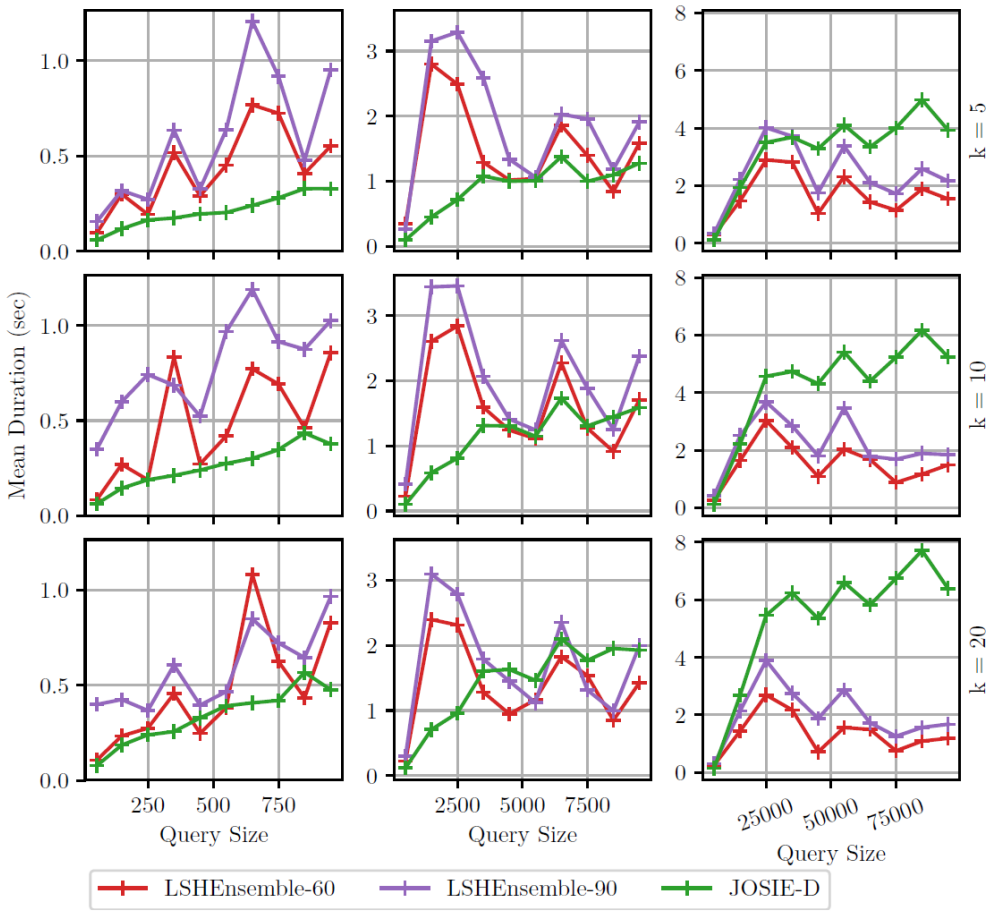
Mean query time on the OpenData benchmark



Mean query time on the WebTable benchmark

The “D” suffix denotes a further optimization to consider distinct posting lists

JOSIE vs LSH Ensemble



Mean query time on the OpenData benchmark

LSH Ensemble with minimum recall of 60% and 90% for the top-k candidates)

JOSIE outperforms LSH Ensemble (*approximate*):

- For **smaller k** (prefix filter more selective than LSH, i.e., very few posting lists and less candidates to read)

AND

- For **smaller queries** (LSH Ensemble wastes time on false positive candidates)

LSH Ensemble outperforms JOSIE if the user wants to **retrieve many results** (k ≥ 20) and **the query size is large** (more than 10k)

MATE: Multi-Attribute Table Extraction

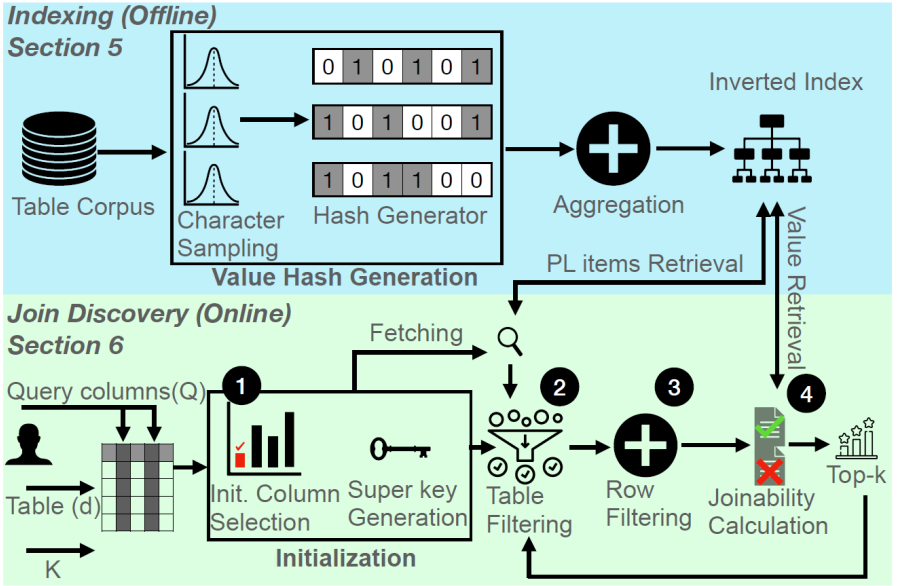
- n-ary join discovery is hard:
  - Cannot build a posting list for each possible combination of attributes
    - Exponential complexity
    - Order dependent
- MATE extends the posting list with a clever probabilistic data structure that allows to quickly look for candidate n-ary join
  - Offline phase create the posting list
  - Online (i.e., at query time) generate the sketch for the desired composite key

Input table (d)

Row	F. Name (q1)	L. Name (q2)	Country (q3)	Salary
1	Muhammad	Lee	US	60k
2	Ansel	Adams	UK	50k
3	Ansel	Adams	US	400k
4	Muhammad	Lee	Germany	90k
5	Helmut	Newton	Germany	300k

Candidate table (T_1)

Row	Vorname	Nachname	Land	Besetzung
1	Helmut	Newton	Germany	Photographer
2	Muhammad	Lee	US	Dancer
3	Ansel	Adams	UK	Dancer
4	Ansel	Adams	US	Photographer
5	Muhammad	Ali	US	Boxer
6	Muhammad	Lee	Germany	Birder
7	Gretchen	Lee	Germany	Artist
8	Adam	Sandler	US	Actor



$$v_i \mapsto \{(T_{i1},\ C_{i1},\ R_{i1},\ S_{i1}), (T_{i2},\ C_{i2},\ R_{i2},\ S_{i2}),\ ...\}$$

Super key

new posting list

Super-key generation

- Super-key:
  - order-independent
  - fixed-sized hash-value
  - merges all possible key values into a single index element

- A possible solution is to use Bloom Filter (BF)
  1. Create a BF for each row
  2. Insert all the attribute values in the BF

What is Bloom Filter? A Compact Data Structure Storing Set-membership

- Bloom Filters answer "is item x in set Y" by:
  - "definitely no", or
  - "probably yes" with probability ε to be wrong

- Benefit: not always precise but highly compact
  - Typically a few bits per item
  - Achieving lower ε (more accurate) requires spending more bits per item

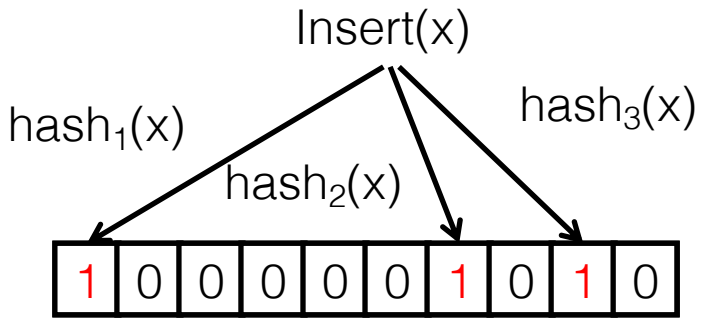
Bloom Filter Example

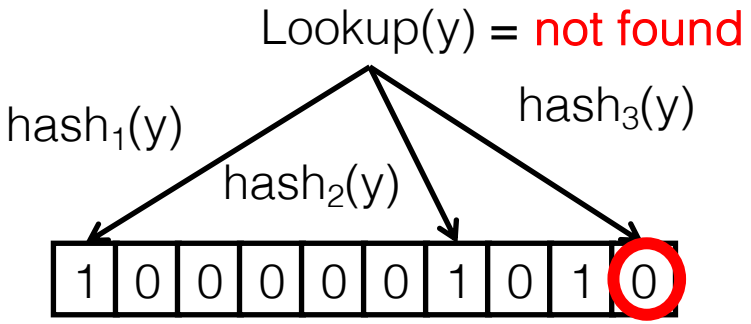
- Anti-spam system:
  - Keep a list of *good* email addresses
    - If an email is in the list, allow the mail
    - We want to **avoid false negative**:
      - User's mail should not be lost
    - We can **tolerate a few false positive**:
      - If we get **some spam, time to time, it's ok**
  - There might be Millions of *good* email address
    - Fast lookup is challenging
- Other examples:
  - Avoid disk lookup in DBMS (check is a tuple is there)
  - join in (distributed) DBMS (send only needed tuples)

Bloom Filter Basics

A Bloom Filter consists of m bits and k hash functions

Example: $m = 10$, $k = 3$



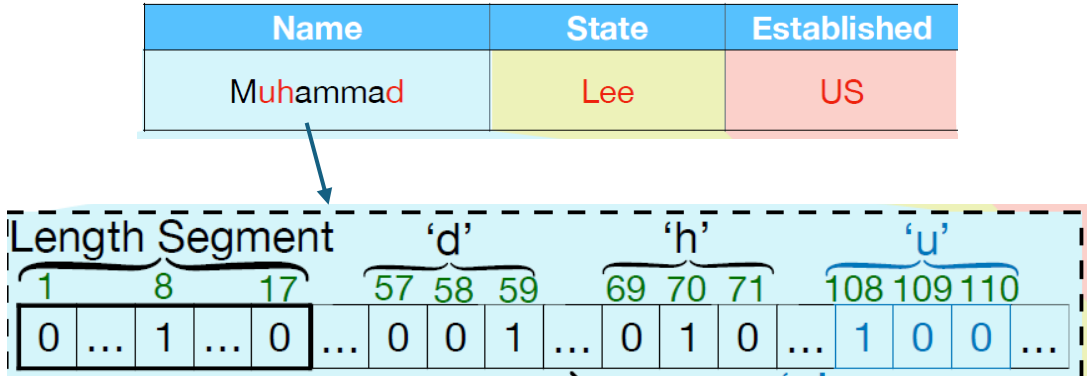


XASH

- BF:
  - employs hash functions that assume a uniform distribution
  - BF is agnostic to the distinguishing properties within columns

- XASH:
  - Probabilistic data structure (as BF)
  - aims at minimizing the 1s to decrease the chances of collisions for super-keys
  - encodes syntactic features into distinguishable hash values
    - Least frequent characters
    - Their location
    - Value length
  - **Significantly reduce the false positive rate compared to BF**

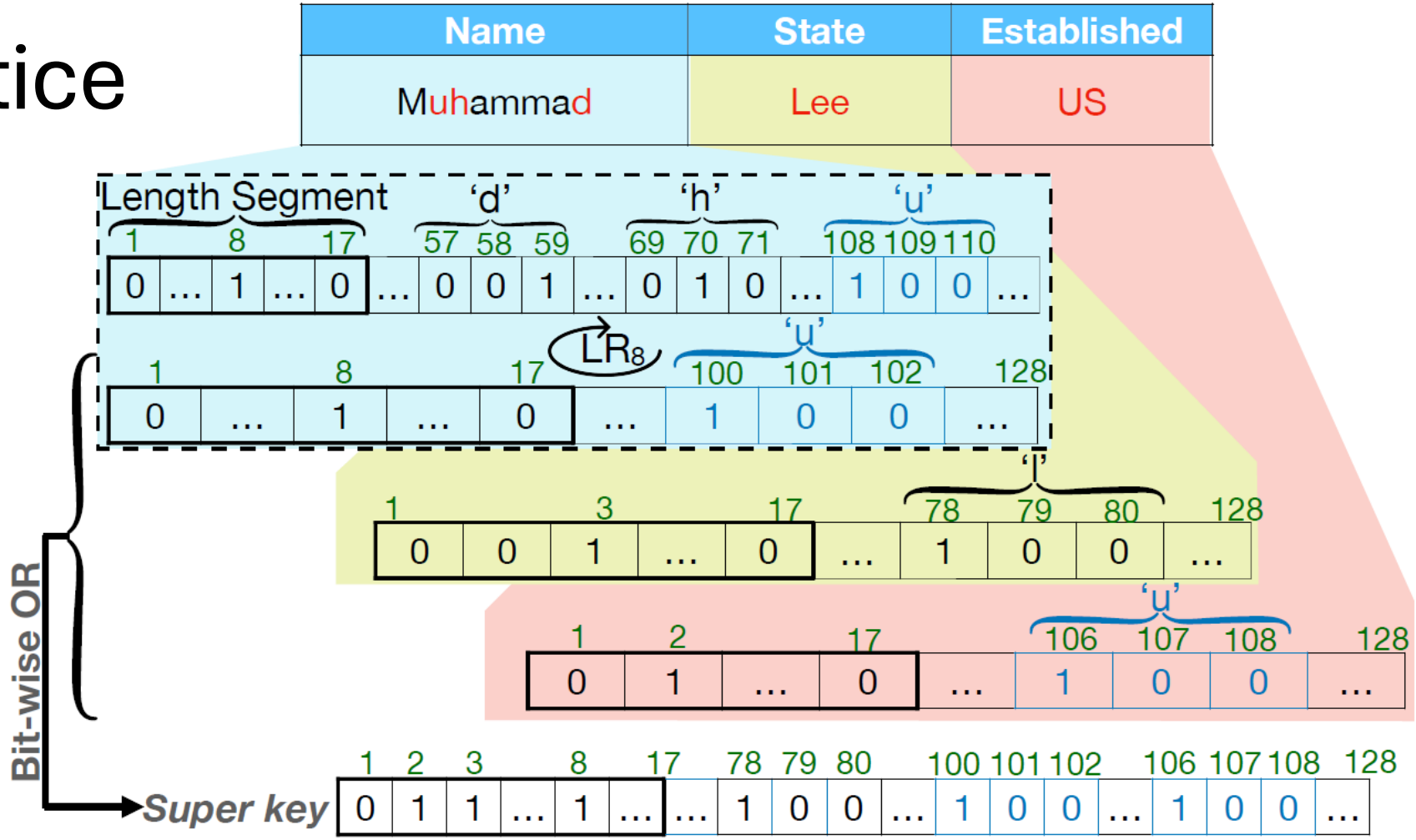
XASH in practice



- Let's consider"
  - 128 bit signature for each super-key
  - 37 symbols for the vocabulary (letters+numbers)
  - 17 maximum length of the attribute value
- For each attribute value we have a partial signature with:
  - 17 bit for the length
    - Only one at a time can be set to represent the lenth (minimize 1s)
    - When checking for super-key match, MATE can use these fist 17 bits to filter
      - If the length of the value is different, no need to check further
  - 111 bit for the 37 symbols
    - 3 bit per symbol
      - 100 if the symbol appears in the first third of the attribute value
      - 010 if the symbol appears in the second third of the attribute value
      - 001 if the symbol appears in the last third of the attribute value

XASH in practice

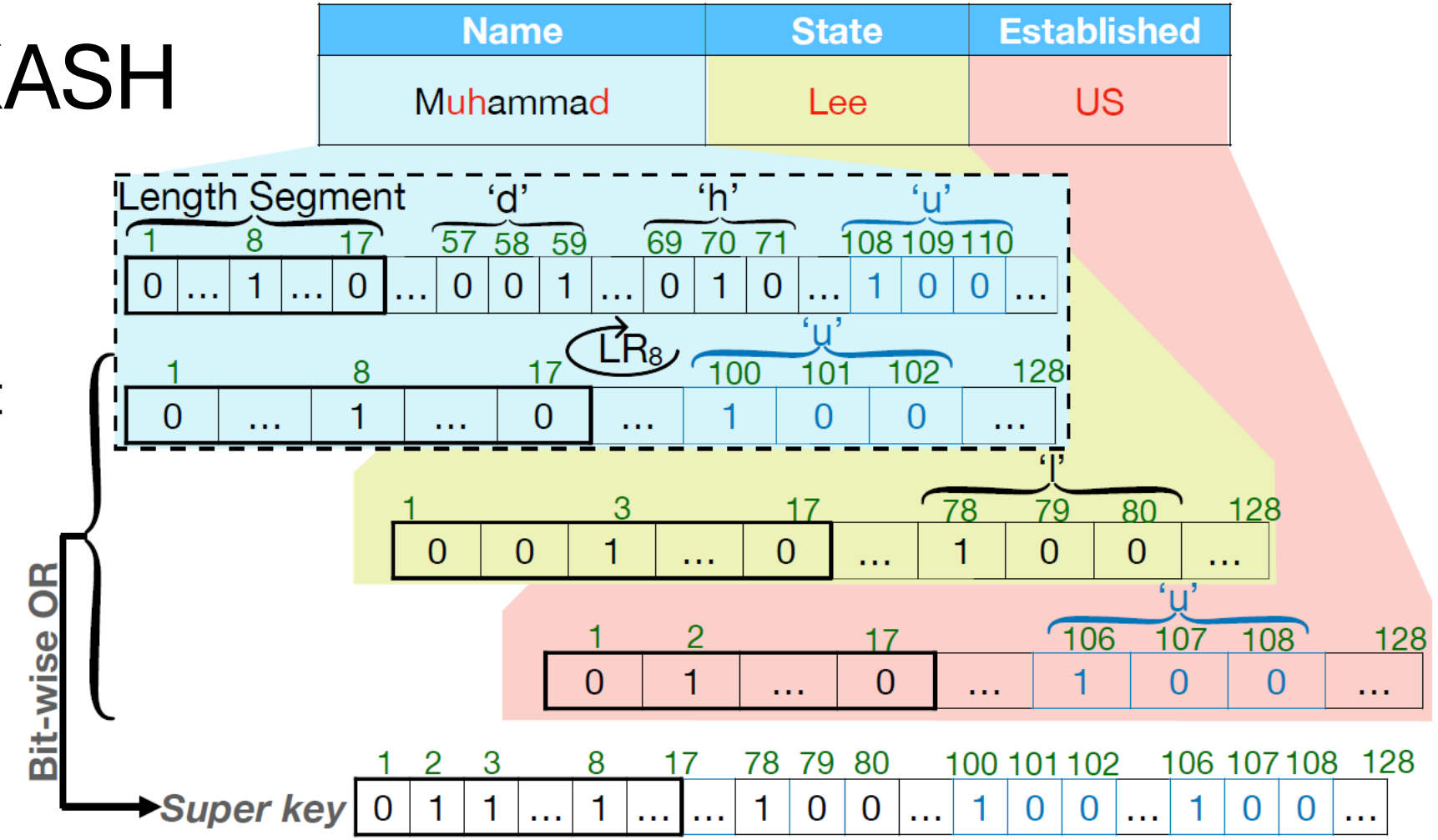
- The 111 bit values are “rotated” to the left of L bits
  - L is the length of the value
  - left-most bits go to the tail
- Proved to reduce the collisions
  - Recuce false positives



How to use XASH

At query time, the XASH of each row-key (the desired set of attribute values) is compared with the super keys:

- If it is not a subset of the super key, it cannot be a join



$T_{XASH} = Q_{XASH} \wedge S_{XASH}$
$Res = SUM(T_{XASH} \text{ XOR } Q_{XASH})$
If Res == 0 → coul be a join

XASH: number of bits

XASH encodes each value into a fixed-size bit array *a*

- Hash values of different individual strings must be different → use all possible bits to encode as many value as possible
- Avoid masking FPs → contain as few *1-bits* as possible (need for an **upper bound**)

Optimal number α of *1-bits* for each hash value generated by XASH (a):

$$\arg\min_{\alpha} \binom{|a|}{\alpha} > C_{unique}$$

Hash size

Unique values in the corpus

Possible combinations of size α over $|a|$ bits

1 *1-bit* reserved to encode the value length,
$\alpha - 1$ for the actual value (characters)

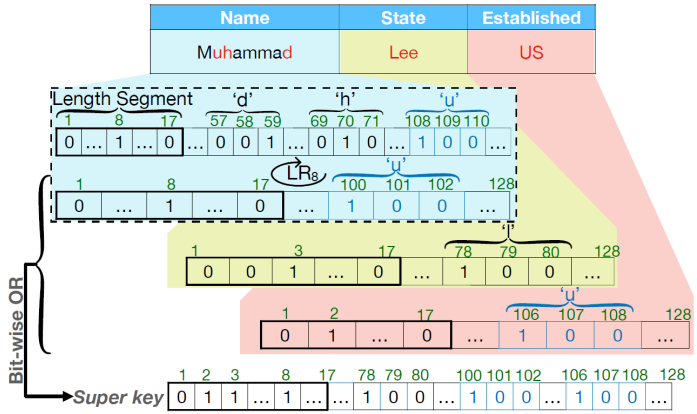
In the example, $|a| = 128$ and $\alpha = 4$
(3 1-bits for the value, 1 for its length)

Input table (d)

Row	F. Name (q1)	L. Name (q2)	Country (q3)	Salary
1	Muhammad	Lee	US	60k
2	Ansel	Adams	UK	50k
3	Ansel	Adams	US	400k
4	Muhammad	Lee	Germany	90k
5	Helmut	Newton	Germany	300k

Candidate table (T_i)

Row	Vorname	Nachname	Land	Besetzung
1	Helmut	Newton	Germany	Photographer
2	Muhammad	Lee	US	Dancer
3	Ansel	Adams	UK	Dancer
4	Ansel	Adams	US	Photographer
5	Muhammad	Ali	US	Boxer
6	Muhammad	Lee	Germany	Birder
7	Gretchen	Lee	Germany	Artist
8	Adam	Sandler	US	Actor



XASH: encoding the characters

$\alpha - 1$ *1-bits* to encode the actual value (characters) – *In the example,* $\alpha - 1 = 3$

Different values should use **different bit segments** of *a*, hence the characters should be maximally different across words → pick the $\boldsymbol{\alpha - 1}$ **least frequent characters** inside a word as the differentiator (minimize collisions)

Depending on the **number of possible characters**, split *a* into as many smaller fixed-size segments, one for each character

Considering all 37 alphanumeric characters (including space), 37 segments of β bits each

$$\arg\max_{\beta} (37 \cdot \beta < |a|)$$

The rest of the array *a* (a_l) is used to encode the value length (using 1 *1-bit*)

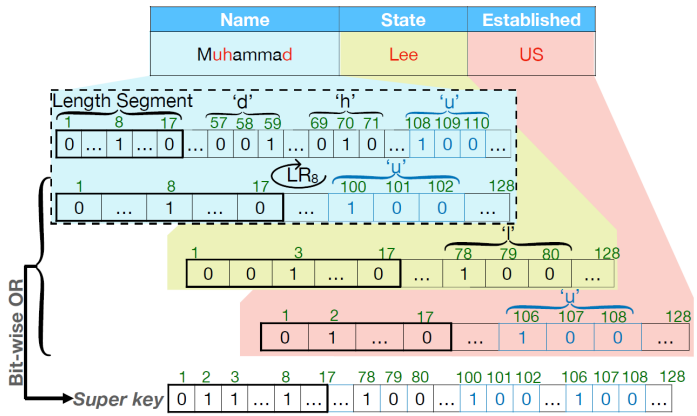
In the example, $|a| = 128$*, hence* $\beta = 3$ *(i.e., 3 bits for each of the 37 characters, 1 1-bit for the 3 least frequent ones) and* $|a_l| = 17$

Input table (d)

Row	F. Name (q1)	L. Name (q2)	Country (q3)	Salary
1	Muhammad	Lee	US	60k
2	Ansel	Adams	UK	50k
3	Ansel	Adams	US	400k
4	Muhammad	Lee	Germany	90k
5	Helmut	Newton	Germany	300k

Candidate table (T_1)

Row	Vorname	Nachname	Land	Besetzung
1	Helmut	Newton	Germany	Photographer
2	Muhammad	Lee	US	Dancer
3	Ansel	Adams	UK	Dancer
4	Ansel	Adams	US	Photographer
5	Muhammad	Ali	US	Boxer
6	Muhammad	Lee	Germany	Birder
7	Gretchen	Lee	Germany	Artist
8	Adam	Sandler	US	Actor



XASH: encoding the character location

If more bits are available for each character (i.e., $\beta > 1$), use them to encode the relative position inside the original string (value) to further distinguish the hash values

- Divide the string into β equal areas from left to right
- Check in which area the character appears
- Set to 1 the corresponding bit in the reserved character segment in *a*

Bit index in the character segment (x):

$$x = \left\lceil \frac{\lambda \cdot \beta}{l_v} \right\rceil$$

Location of the character to encode (average of the locations in case of repetition)

Number of characters (length) of the value

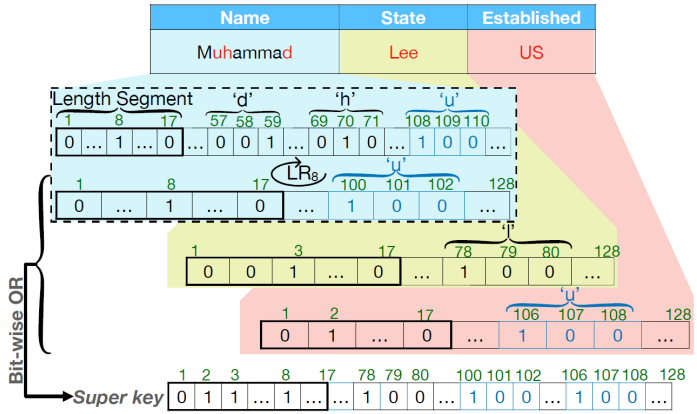
In the example, $\beta = 3$

Input table (d)

Row	F. Name (q1)	L. Name (q2)	Country (q3)	Salary
1	Muhammad	Lee	US	60k
2	Ansel	Adams	UK	50k
3	Ansel	Adams	US	400k
4	Muhammad	Lee	Germany	90k
5	Helmut	Newton	Germany	300k

Candidate table (T_1)

Row	Vorname	Nachname	Land	Besetzung
1	Helmut	Newton	Germany	Photographer
2	Muhammad	Lee	US	Dancer
3	Ansel	Adams	UK	Dancer
4	Ansel	Adams	US	Photographer
5	Muhammad	Ali	US	Boxer
6	Muhammad	Lee	Germany	Birder
7	Gretchen	Lee	Germany	Artist
8	Adam	Sandler	US	Actor



XASH: encoding the length and rotating bits

$|a_l|$ bits used to encode the length of the value l_v (using 1 *1-bit*)

Set to one the bit of a_l corresponding to l_v mod $|a_l|$

The length segment is positioned as the left-most one → optimization to skip unnecessary bit operations (if no value in the super key of the candidate row has the same length as a query value, no need to check character segments)

Random matches: when a key value partially masked by the hash function of individual values is masked by the aggregated super key

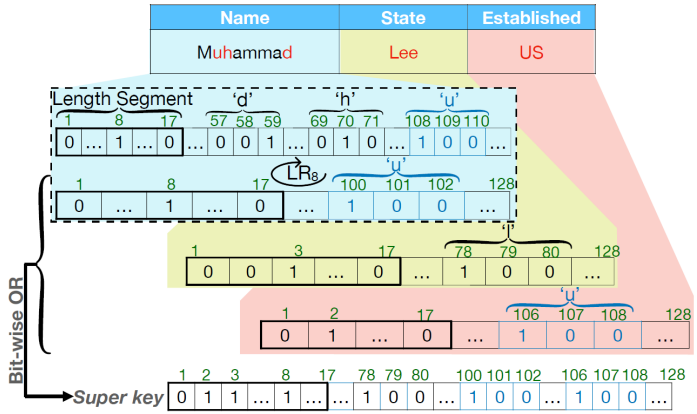
To further reduce their likelihood, rotate the character segments of a value to the left by its length l_v

Input table (d)

Row	F. Name (q1)	L. Name (q2)	Country (q3)	Salary
1	Muhammad	Lee	US	60k
2	Ansel	Adams	UK	50k
3	Ansel	Adams	US	400k
4	Muhammad	Lee	Germany	90k
5	Helmut	Newton	Germany	300k

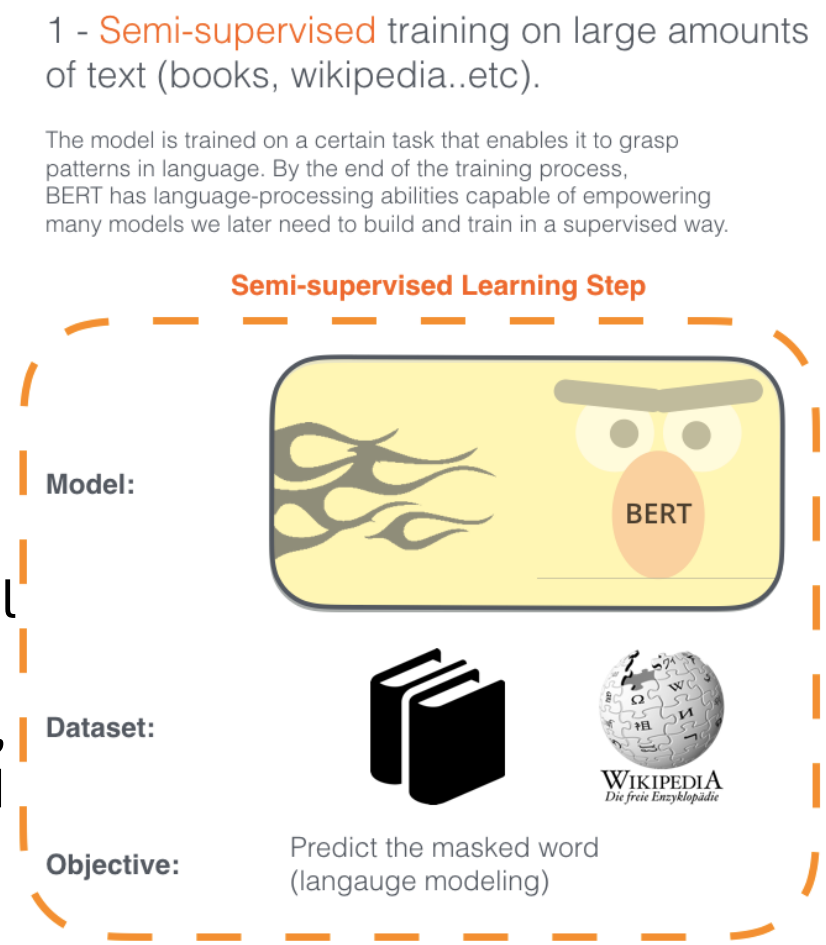
Candidate table (T_1)

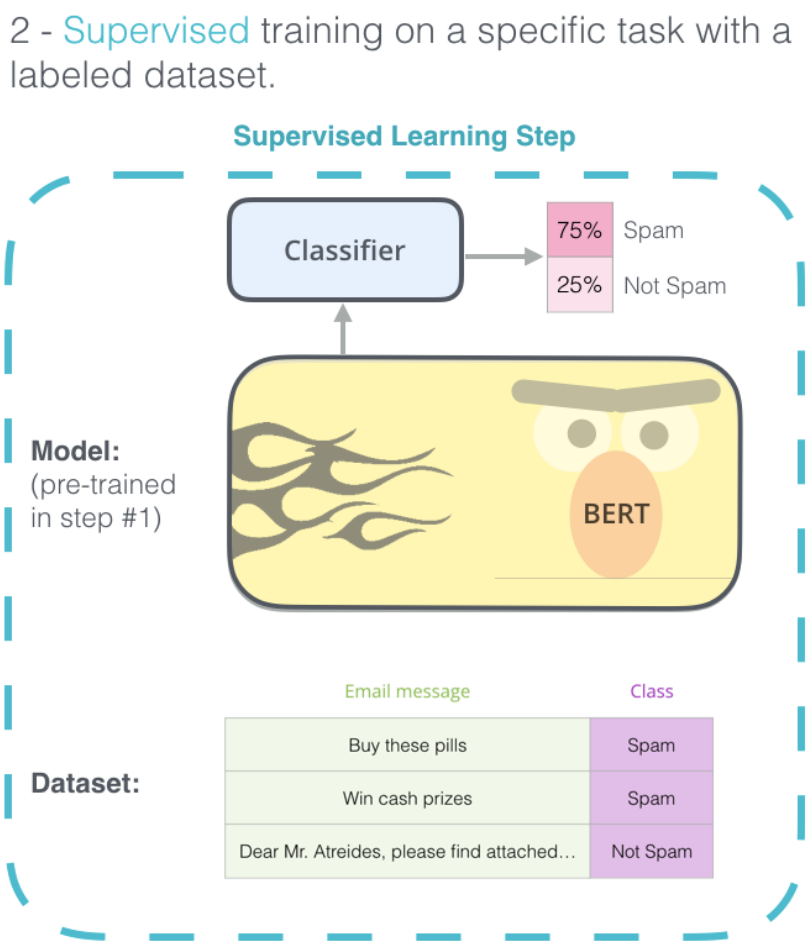
Row	Vorname	Nachname	Land	Besetzung
1	Helmut	Newton	Germany	Photographer
2	Muhammad	Lee	US	Dancer
3	Ansel	Adams	UK	Dancer
4	Ansel	Adams	US	Photographer
5	Muhammad	Ali	US	Boxer
6	Muhammad	Lee	Germany	Birder
7	Gretchen	Lee	Germany	Artist
8	Adam	Sandler	US	Actor



Transformer background

- BERT (based on the transformer architecture)
  - pre-trained on massive datasets
  - can handle language-based tasks
  - allow to build ML model involving language processing saving time, energy, knowledge, and resources
    - **fine-tuning**





http://jalammar.github.io/illustrated-bert/

Example of use

1. Special token [CLS] to get the embedding of the input sentence
2. Feed it to a single-layer neural network as the classifier
3. Train on spam/not-smap
4. Use the Model for inference

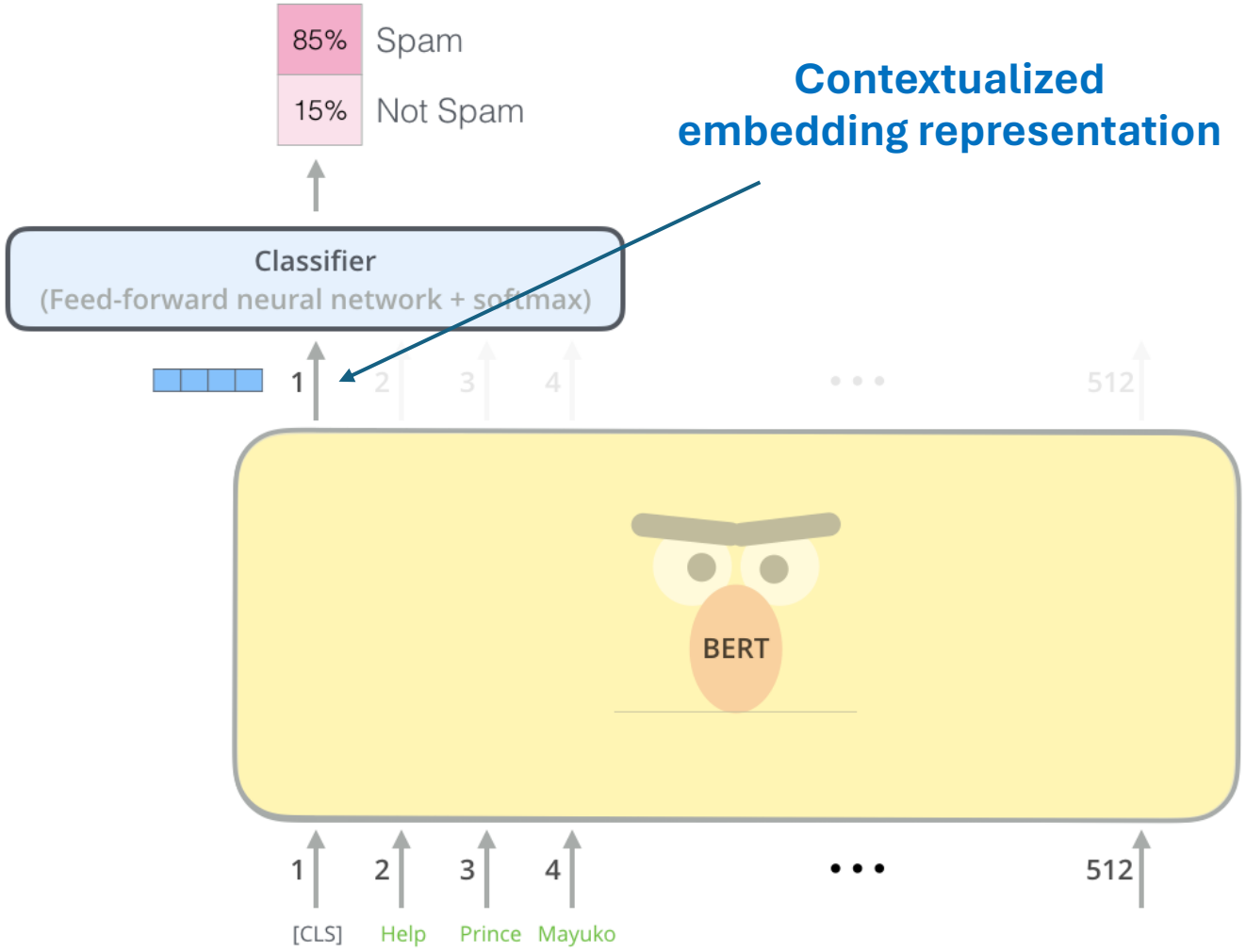


Table understanding

- BERT (as similar models) is very good at capturing the semantic of the language → Language understanding

- Can we to that for table?
  - **Column** understanding: Semantic Type Detection, Domain Discovery
  - **Column-pair** understanding: Relation Extraction
  - **Row** understanding: Row-level entity linking
  - **Table** understanding: Table-level entity linking, topic modelling

- Naïve way:
  - Serialize tables:
    - [ROW][COL] Player [VAL] Zio Williamson [COL]…
    - Player | Zio Williamson | Pos | PF |…; Player |

Player	Pos.	Nationality[n 1]	Team
Zion Williamson+	PF	United States	New Orleans Pelicans
Ja Morant*~	PG	United States	Memphis Grizzlies
RJ Barrett	SG/SF	Canada	New York Knicks
De'Andre Hunter	SF	United States	Los Angeles Lakers (traded to Atlanta via New Orleans)[i][ii]

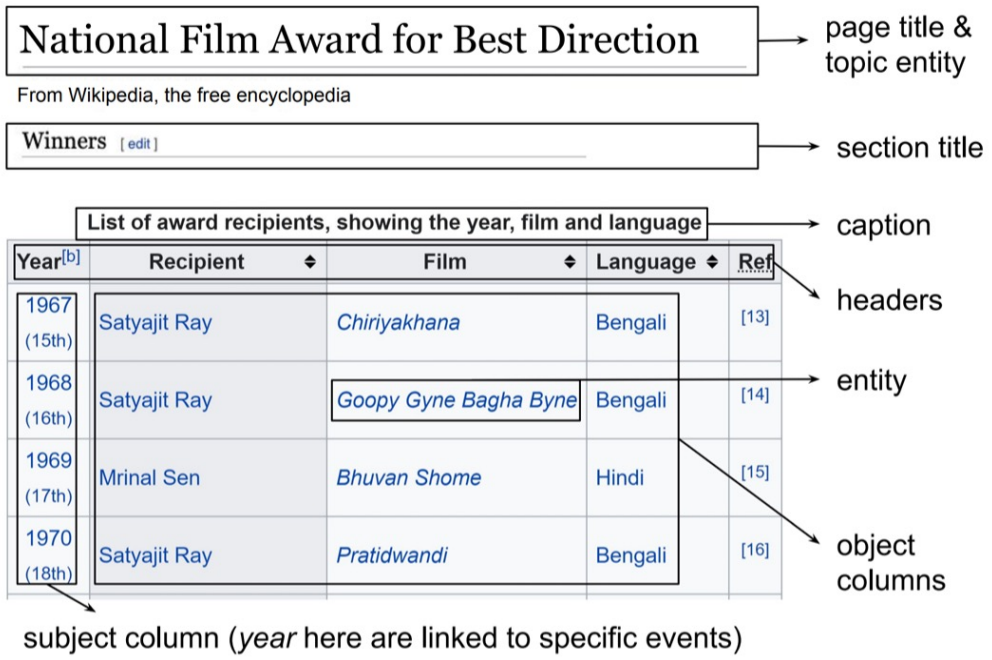
TURL Intuition

- **T**able **U**nderstanding through **R**epresentation **L**earning
  - Adopt the **pre-training/fine-tuning** paradigm (e.g., BERT [1])
    1. Pre-training on large-scale unsupervised data
    2. Fine-tuning on downstream tasks with task-specific supervision
  - No need for expensive feature engineering
  - Requires less (or none) labelled data

- **Contextualized representations**: consider varied use of words/entities in different contexts

[1] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova: "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". Proceedings of the Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), 2019. https://doi.org/10.18653/v1/n19-1423

Table components



Relational Web table $T = (E, C, H, e_t)$

- Columns E containing entities
  - Each entity cell $e \in E$ contains a specific object with a unique identifier
  - $e = \{e^e, e^m\}$, where e^e is the specific entity linked to the cell and e^m is the entity mention (i.e., the text string)
- Caption C: short description (sequence of tokens)
- Schema $H = \{h_0 \dots h_m\}$: headers
- Topic entity $e_t = \{e_t^e, e_t^m\}$: what the table is about (usually extracted from caption or page title)

Learn in an **unsupervised** manner a **task-agnostic contextualized vector representation** for each token in C and H and for each entity (i.e., for e_t and for every $e \in E$)

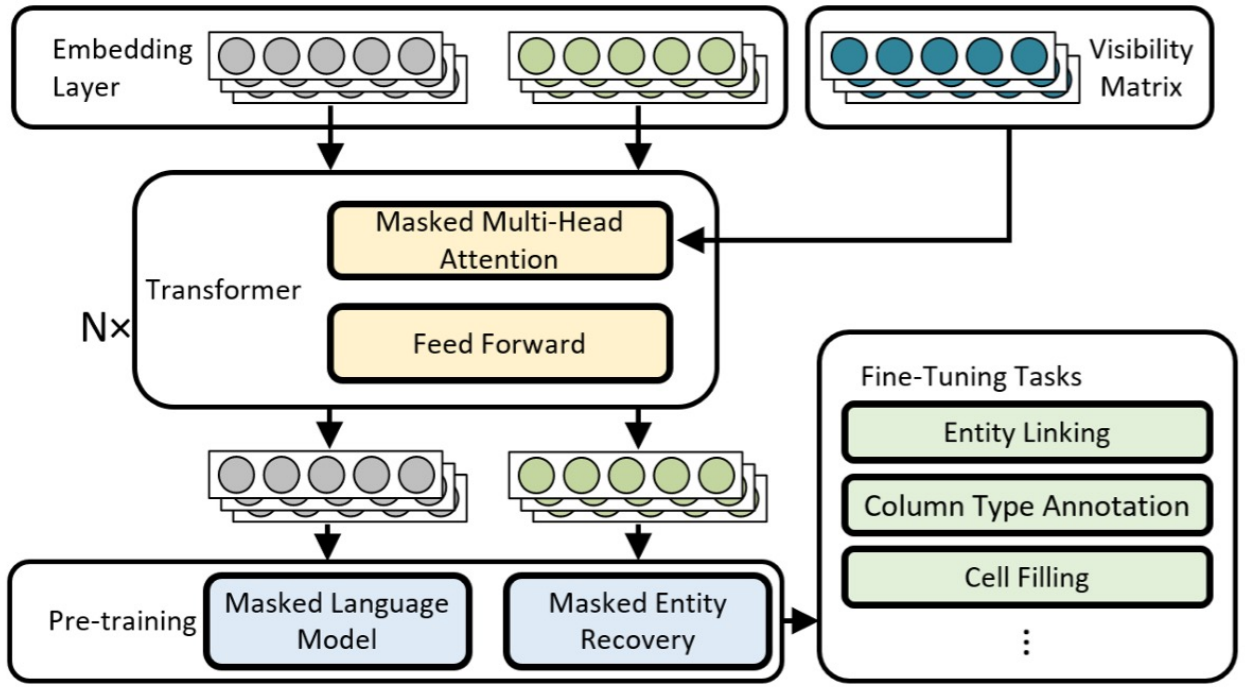
TURL Model

Learning **deep contextualized representations** on relational tables via:

- **pre-training** in an **unsupervised** manner
- and **task-specific fine-tuning**

Two main challenges:

1. **relational table encoding**
2. and **factual knowledge modeling**

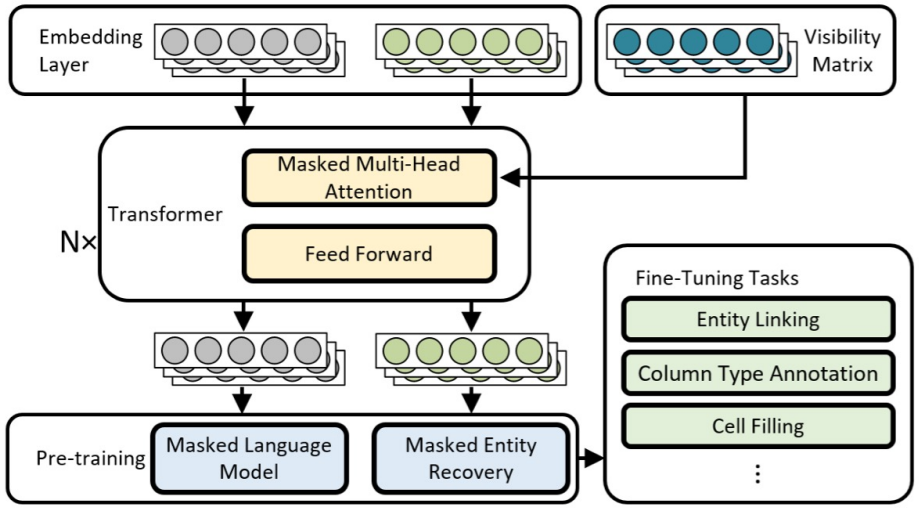


Relational table encoding

Data in relational tables: **semi-structured format**

Model the **row-and-column structure** and integrate the **heterogeneous metadata** (header, captions, links)

- Encode information from different components into **separate input embeddings** and **fuse them together**
- **Structure-aware** **transformer encoder** with **masked self-attention**: each element only aggregates information from structurally related elements (**visibility matrix** based on table structure)

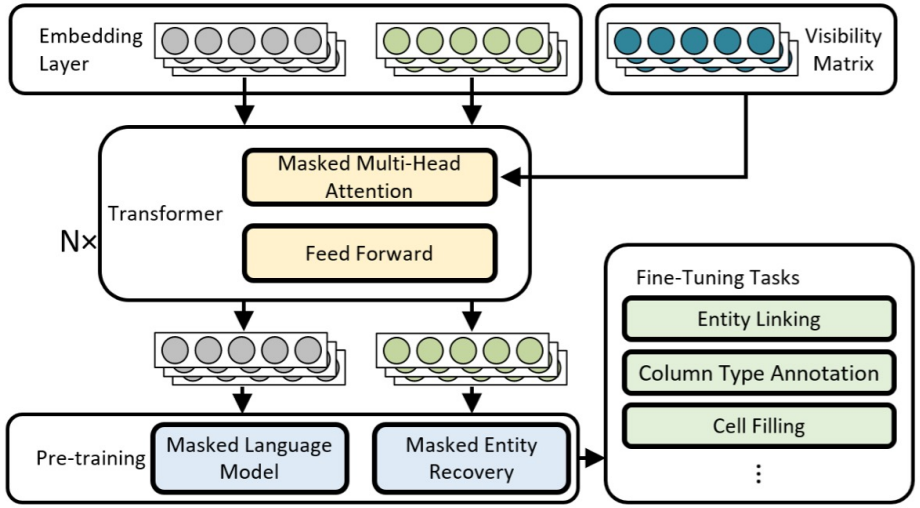


Factual knowledge modeling

Integrate **factual knowledge about entities**, not captured by existing language models directly

Self-supervised training

- Learn **embeddings for each entity** during pre-training
- Model the **relation between entities** in the same row/column (**visibility matrix**)
- **Masked Language Model (MLM)** pre-training objective (from BERT): model the complex characteristics of word use in table metadata
- **Masked Entity Recovery (MER)** pre-training objective:
  - Randomly mask entities in a table to recover them based on other entities and table context → Learn factual knowledge from the tables and encode it into entity embeddings
  - Use the entity mention as additional information → Build connection between words and entities



Embedding layer

Convert different components of an input table $T = (E, C, H, e_t)$ into input embeddings

Linearize the input into a sequence of tokens and entity cells by concatenating the table metadata and scanning the table content row by row

The embedding layer convert each token in C and H and each entity in E and e_t into an embedding representation

Word embedding vector

$$\mathbf{x}^{\mathrm{t}} = \mathbf{w} + \mathbf{t} + \mathbf{p}$$

Position embedding vector

Input token representation

Type embedding vector (in caption or header)

Input entity representation

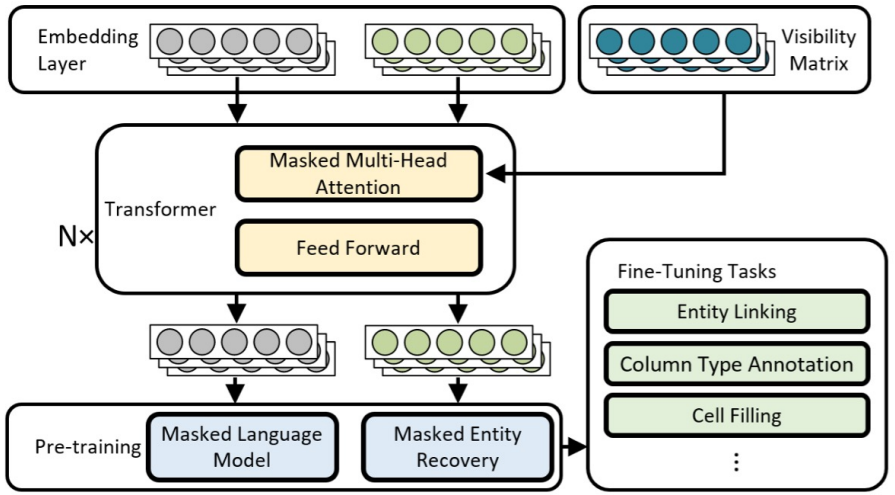
Entity embedding

$$\mathbf{x}^{\mathrm{e}} = \mathtt{LINEAR}([\mathbf{e}^{\mathrm{e}}; \mathbf{e}^{\mathrm{m}}]) + \mathbf{t}^{\mathrm{e}};$$

Type embedding vector (subject/object/topic)

$$\mathbf{e}^{\mathrm{m}} = \mathtt{MEAN}(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_j, \ldots)$$

Average word embedding



A sequence of token and entity representations is fed into the transformer to produce contextualized representations

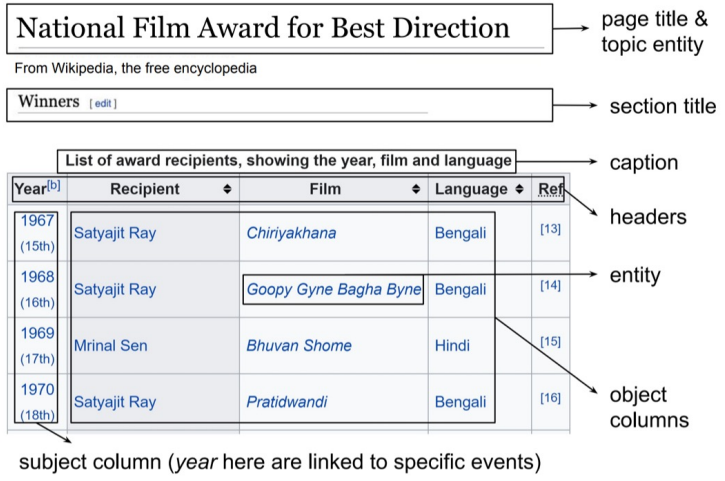
Structure-aware transformer encoder

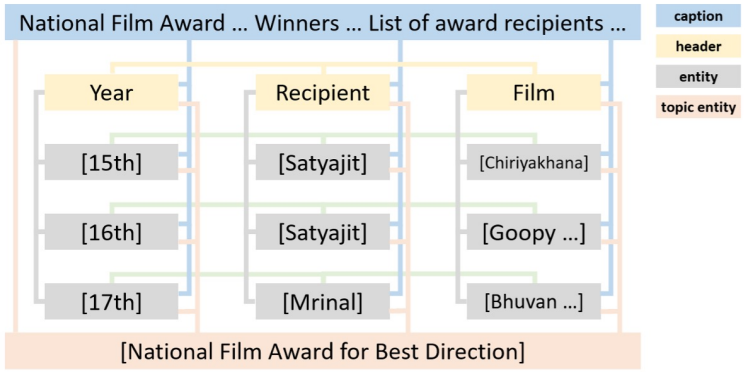
Visibility matrix *M* to model row-column structure

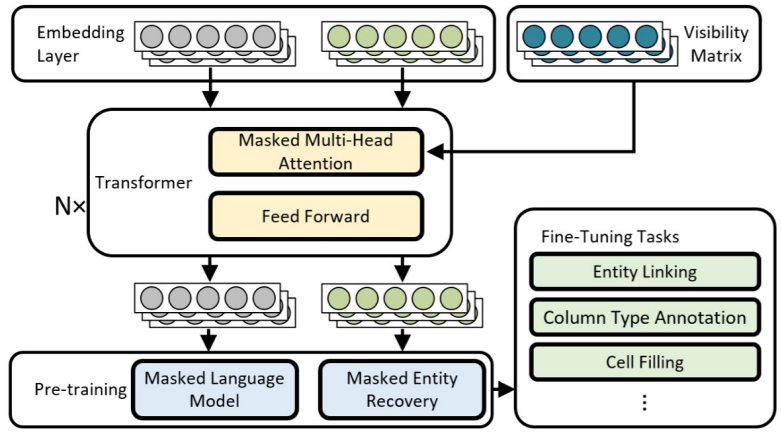
M acts as an **attention mask**: each token/entity can only aggregate information from other structurally related tokens/entities during the self-attention calculation

M is a symmetric binary matrix, with $M_{i,j} = 1$ if and only if $element_j$ is visible to $element_i$

- Caption and topic entity visible to all components
- Entities and text content in the same row or the same columns are visible to each other



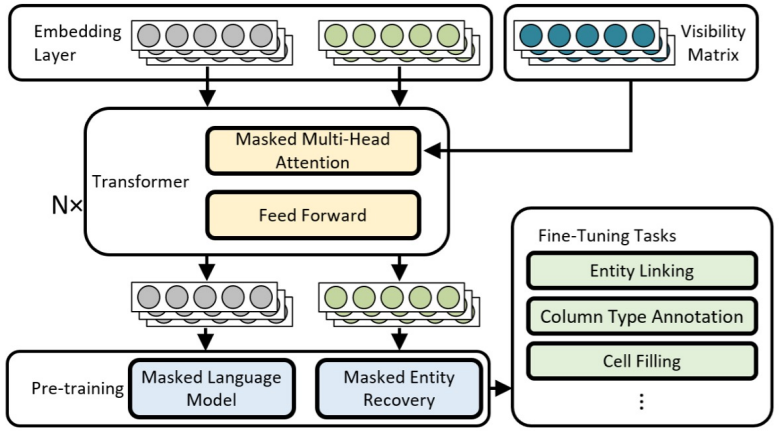


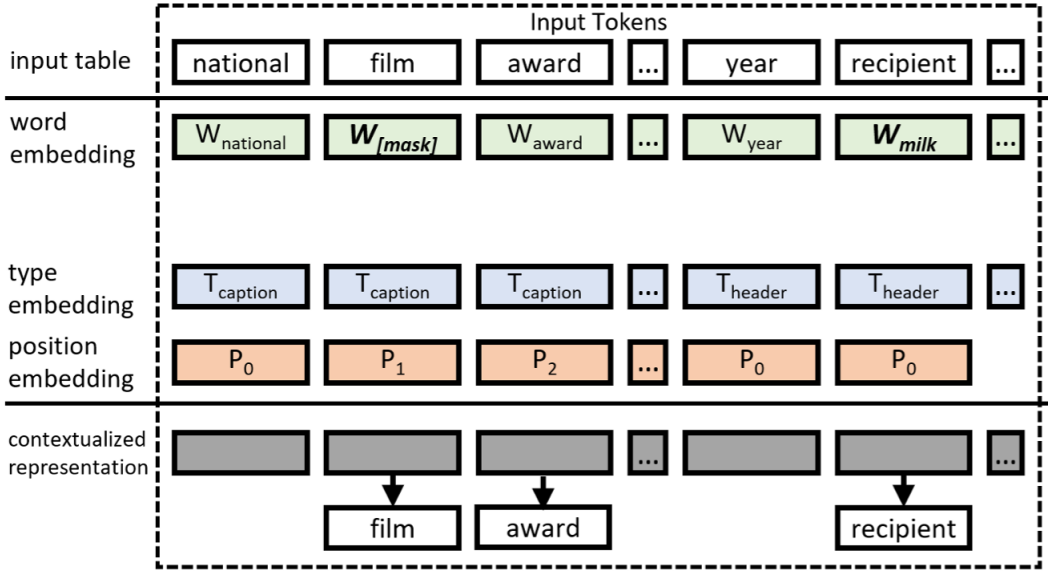


Masked Language Model (MLM) objective

Same as BERT, to **learn representations for tokens**

- Input token sequence (caption and headers)
- Mask some percentage of the tokens (20% positions) at random, then predict such tokens
  - 80% of the time, replace it with a special [MASK] token
  - 10% of the time, replace it with another random token
  - 10% of the time, keep it unchanged

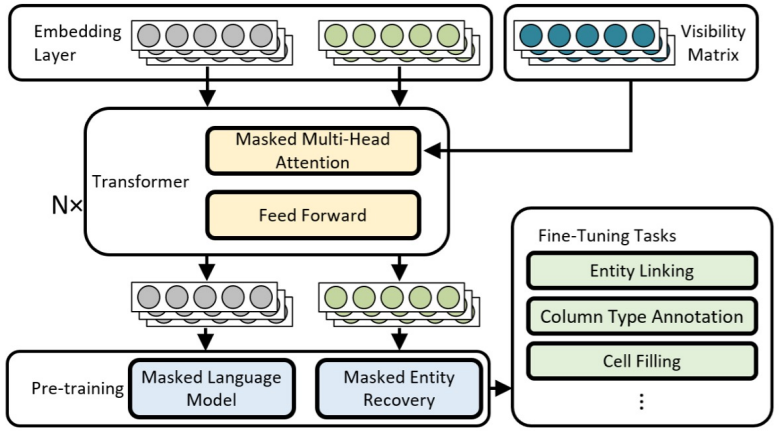


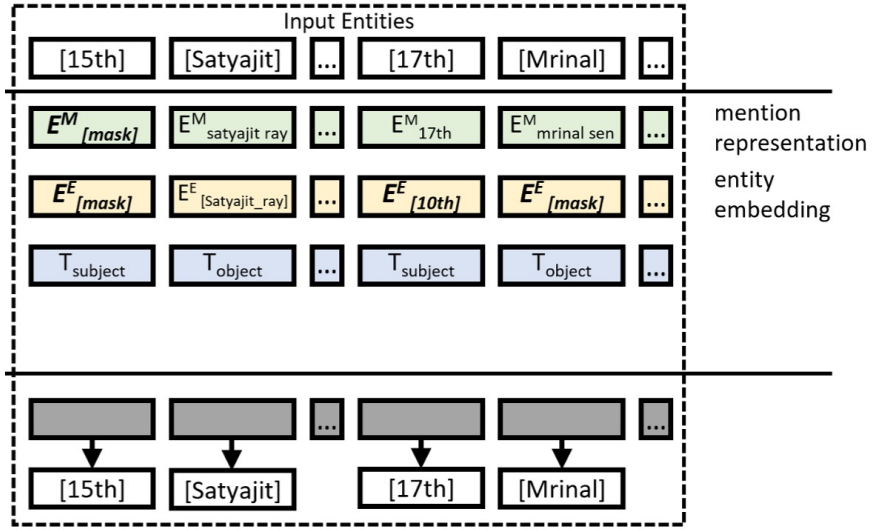


Masked Entity Recovery (MER) objective

To capture the **factual knowledge** embedded in the table content and the **associations between table metadata and table content**

- Mask some percentage (60%) of entity cells at random, then recover the linked entity based on surrounding entity cells and metadata
  - 10% of the time, keep both e^e and e^m unchanged
  - 63% (i.e., 70% of the remaining 90%) of the time, mask both e^e and e^m
  - 27% (i.e., 30% of the remaining 90%) of the time, keep e^m unchanged and mask e^e (10% of the time, replace with random entity to inject noise)

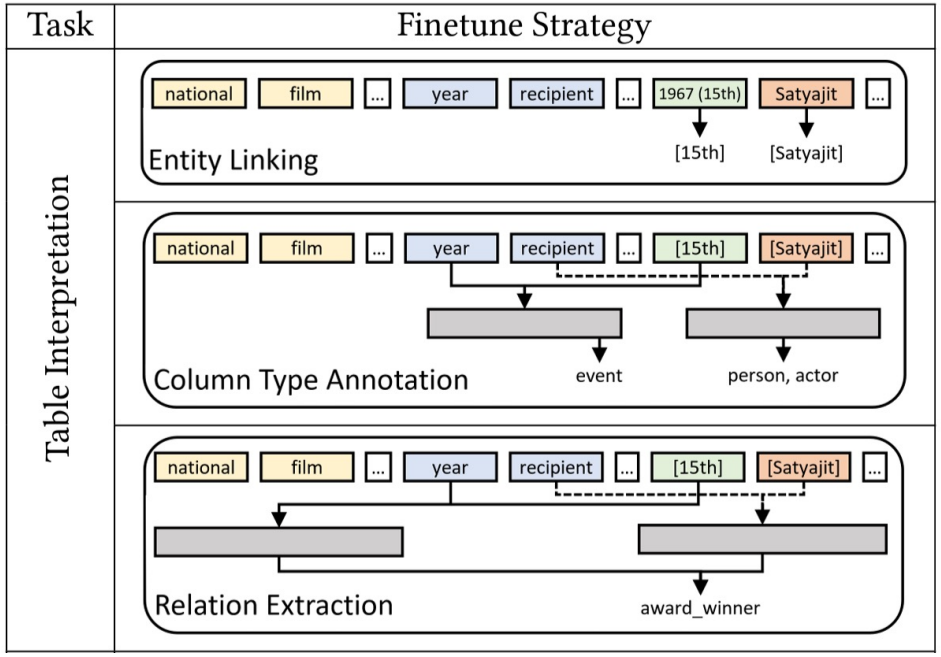


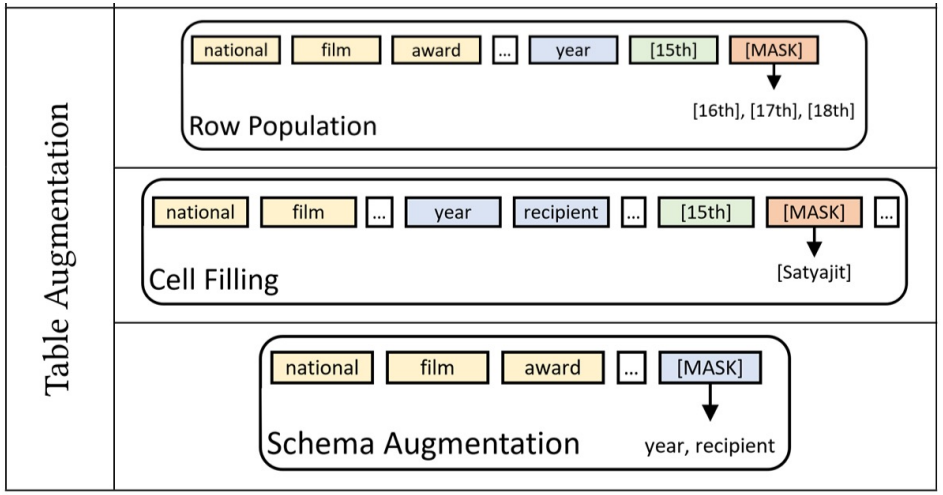


Pre-training and fine-tuning

Pre-train the model on 570K relational tables from the WikiTable corpus [1]

Fine-tuning on 6 downstream tasks





[1] Chandra Sekhar Bhagavatula, Thanapon Noraset, Doug Downey: “TabEL: Entity Linking in Web Tables”. Proceedings of the International Semantic Web Conference (ISWC), 2015. https://doi.org/10.1007/978-3-319-25007-6_25

BY CURTESY OF

Unicorn: A Unified Multi-tasking Model for Supporting Matching Tasks in Data Integration

Jianhong Tu
Renmin University, China

Ju Fan
Renmin University, China

Nan Tang
QCRI/ HKUST (GZ), Qatar / China

Peng Wang
Renmin University, China

Guoliang Li
Tsinghua University, China

Xiaoyong Du
Renmin University, China

Xiaofeng Jia
Beijing Big Data Centre, China

Song Gao
Beijing Big Data Centre, China

Data Matching Tasks

- Data matching generally refers to the process of deciding whether two data elements are the same (a.k.a. a “match”)

Data Elements

String

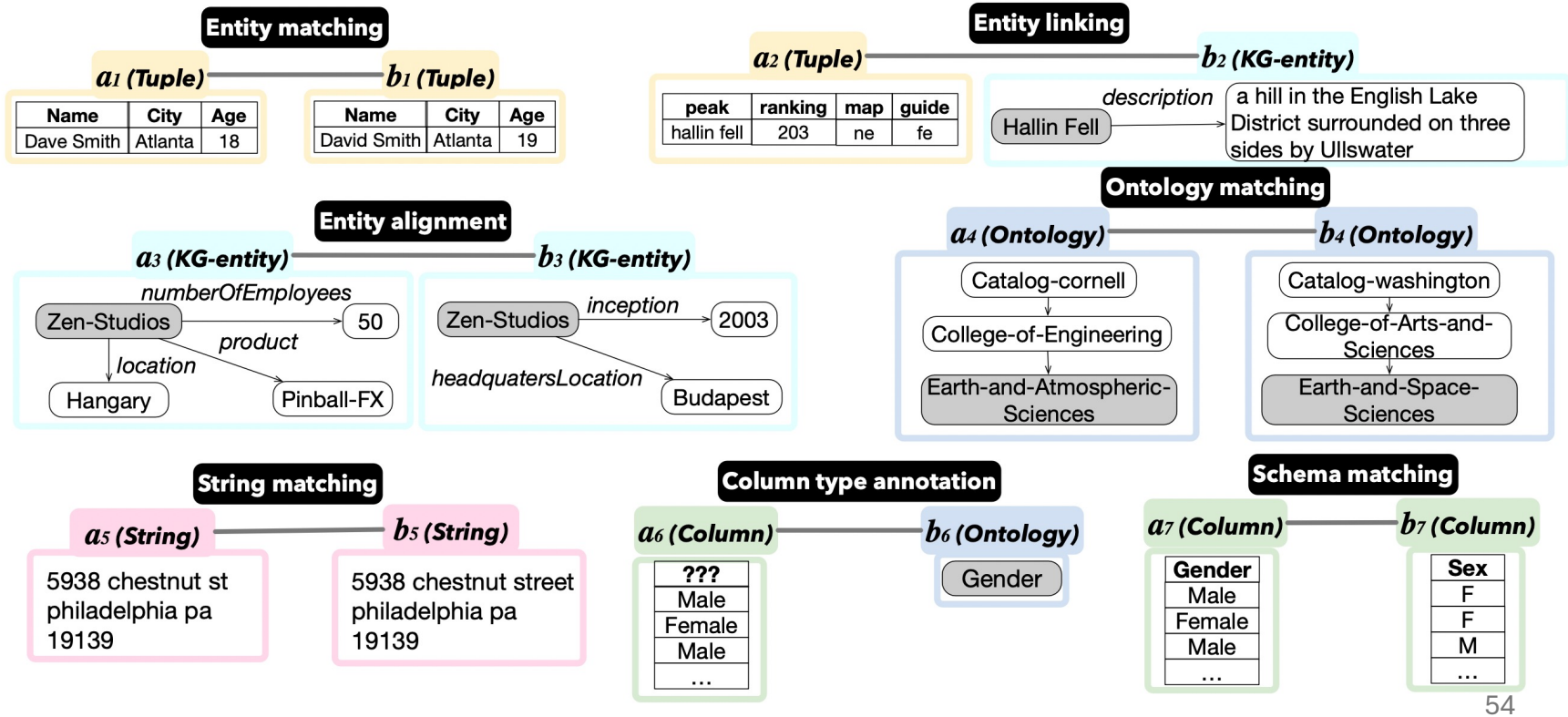
Tuple

Column

Ontology (tree)

Knowledge Graph Entity

Seven Common Data Matching Tasks



Existing Solutions

- Due to their importance, almost all matching tasks have been studied for decades, and remain to be important research topics.
  - DeepMatcher[1] and Ditto[2] for entity matching, Hybrid[3] and TURL[4] for entity linking , HNN+P2Vec[5] for column type annotation, etc.
  - Current solutions are task-specific or even dataset-specific
- Limitations of the specific models
  - The learned knowledge cannot be shared across different models (e.g., EM vs. ColTypeAnn)
    - But they might build latent representations that could help each other
  - One model has to be learned for each task or dataset, which is inefficient and has a high monetary cost

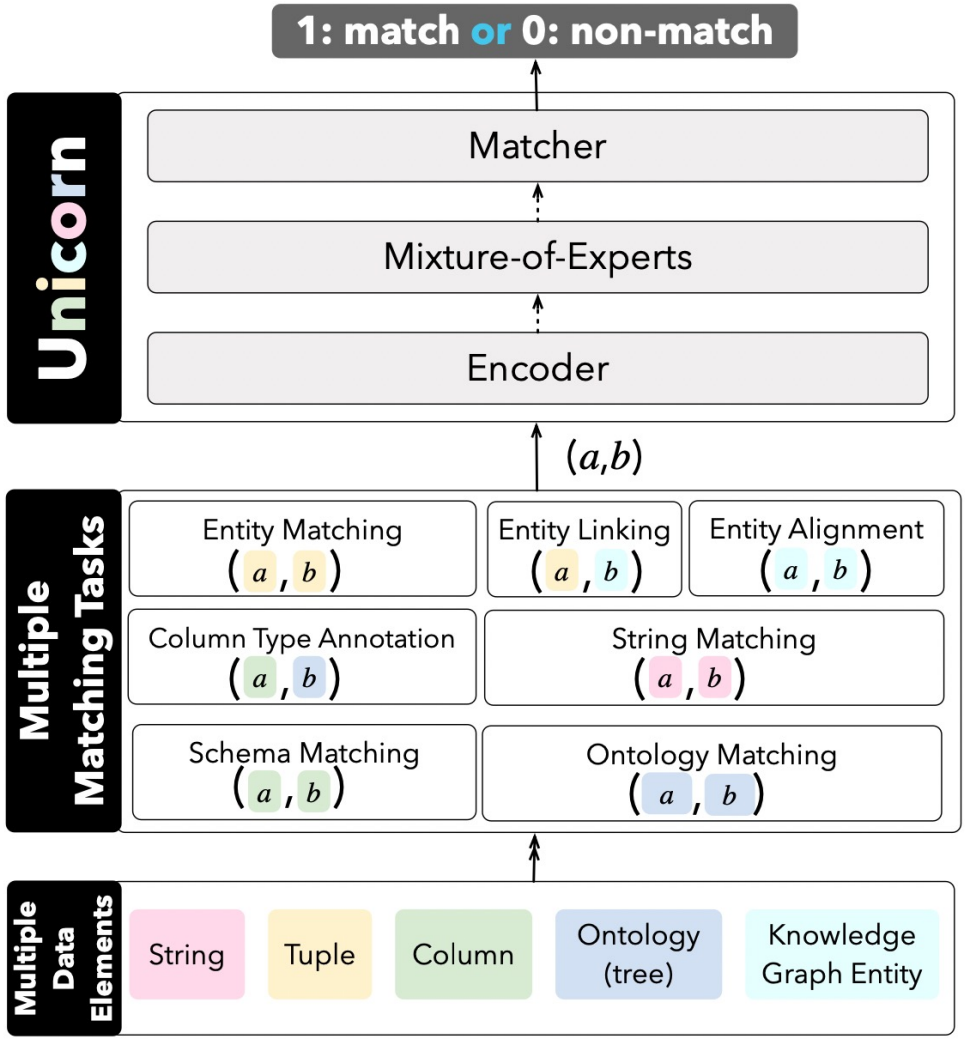
	Task Type	Task	Previous SOTA (Labels)
Model1 →	Entity Matching	DBLP-Scholar	95.6 (22,965)
Model2 →	String Matching	Product	67.18 (1,020)
Model3 →	Entity Alignment	SRPRS: DBP-WD	99.6 (4,500)

[1] Mudgal S, Li H, et al. Deep learning for entity matching: A design space exploration. SIGMOD 2018.
[2] Li Y, Li J, et al. Deep entity matching with pre-trained language models. VLDB 2020.
[3] Efthymiou V, Hassanzadeh O, et al. Matching web tables with knowledge base entities: from entity lookups to entity embeddings. ISWC 2017.
[4] Deng X, Sun H, et al. Turl: Table understanding through representation learning. SIGMOD 2022.
[5] Chen J, Jiménez-Ruiz E, et al. Learning semantic annotations for tabular data. IJCAI 2019.

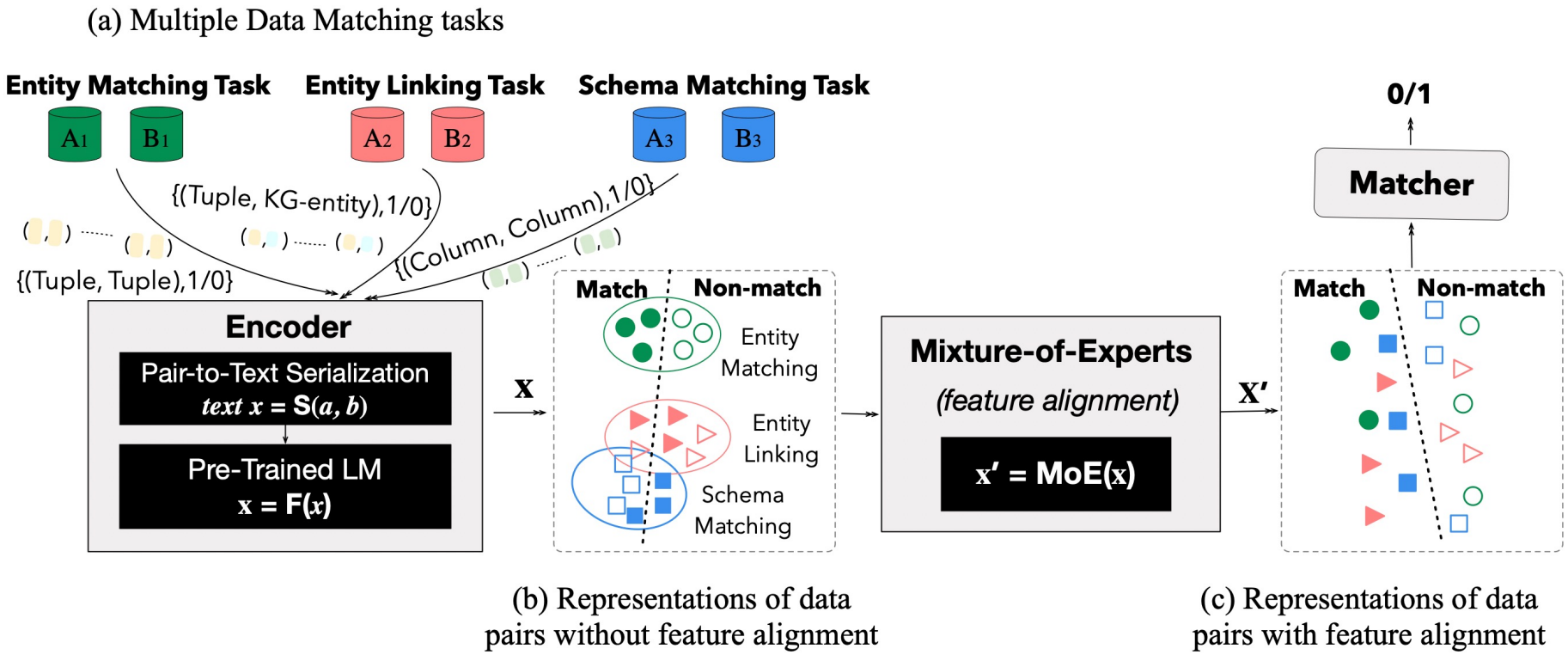
Can we build a unified model that learns from multiple tasks/datasets?

Unicorn: A Unified Model for Data Matching

- Task unification: A unified model to serve a variety of data matching tasks
- Multi-task learning: Enabling knowledge sharing across multiple data matching, which may even outperform specific models
- Zero-shot prediction: Making predictions for a new task or a new dataset with zero labeled matching/non-matching pairs
- Challenges in building such a unified model:
  - **Heterogeneous formats**: Data elements have different data formats
  - **Unique matching semantics**: Tasks have different data matching semantics

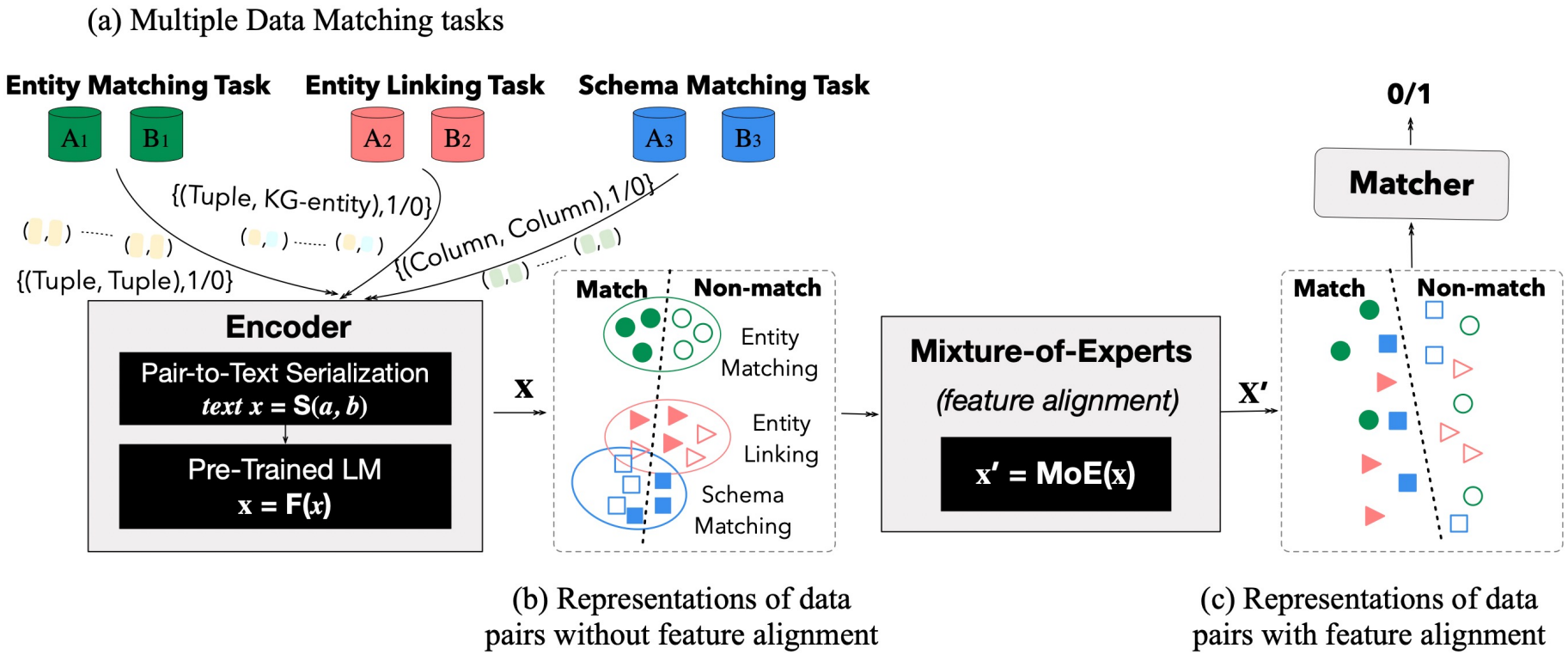


A General Framework of Unicorn



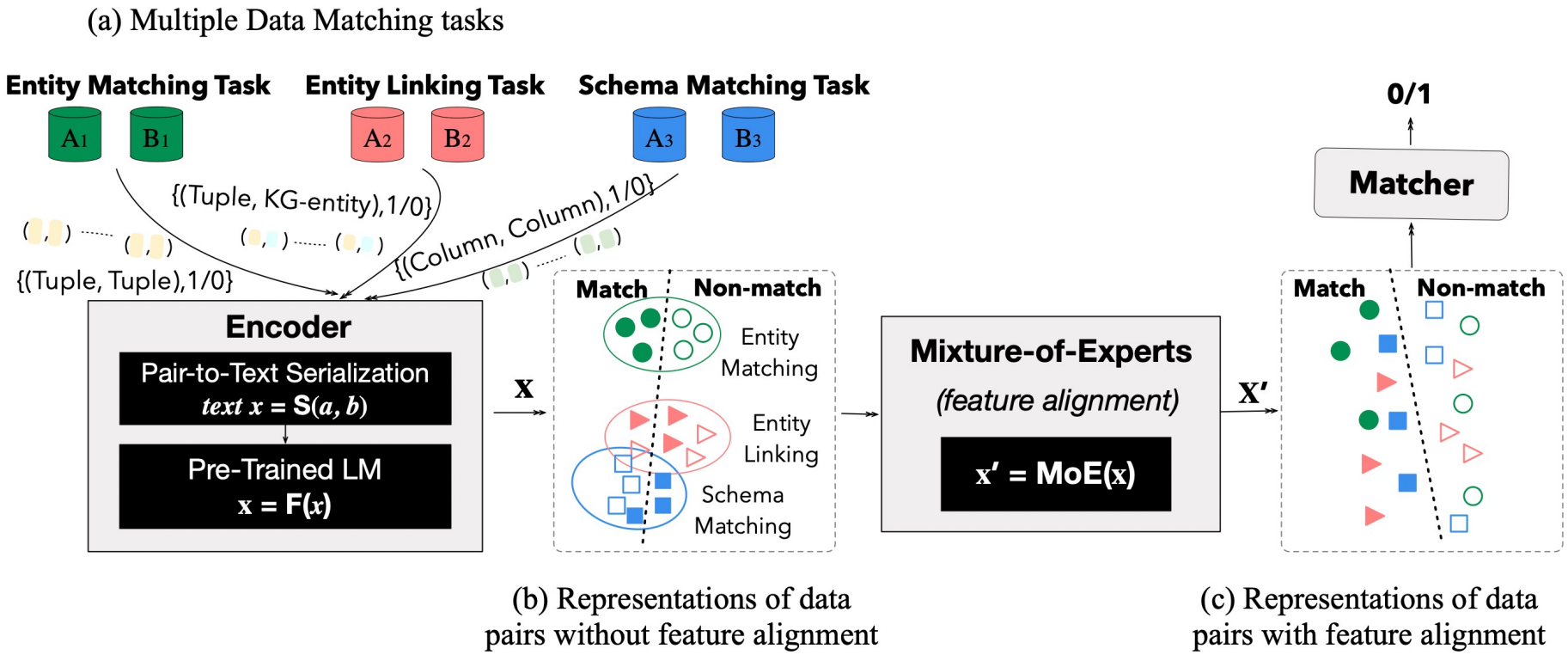
- **Encoder** converts any pair of elements with heterogeneous formats into a learned representation **x** based on Pair-to-Text Serialization and Pre-trained Language Model
  - 🟢, 🔺, 🟦 represent feature vectors of matching/non-matching element for different tasks
  - Although all the pairs from different tasks are mapped into one feature space, the distributions of their representations may **not be aligned**, so, it is **hard to train a good Matcher**

A General Framework of Unicorn



- **Mixture-of-Experts (MoE)** layer enhances the representation **x** into a better representation **x'** with feature alignment
  - The MoE layer enhances the representation x into a better representation x' with feature alignment
  - such that a good Matcher is easier to learn

A General Framework of Unicorn



- **Encoder** converts any pair of elements with heterogeneous formats into a learned representation **x** based on Pair-to-Text Serialization and Pre-trained Language Model
- **Mixture-of-Experts (MoE)** layer enhances the representation **x** into a better representation **x'** with feature alignment
- **Matcher** predicts either 1/0 (match/non-match) by taking the above representation as input

Encoder

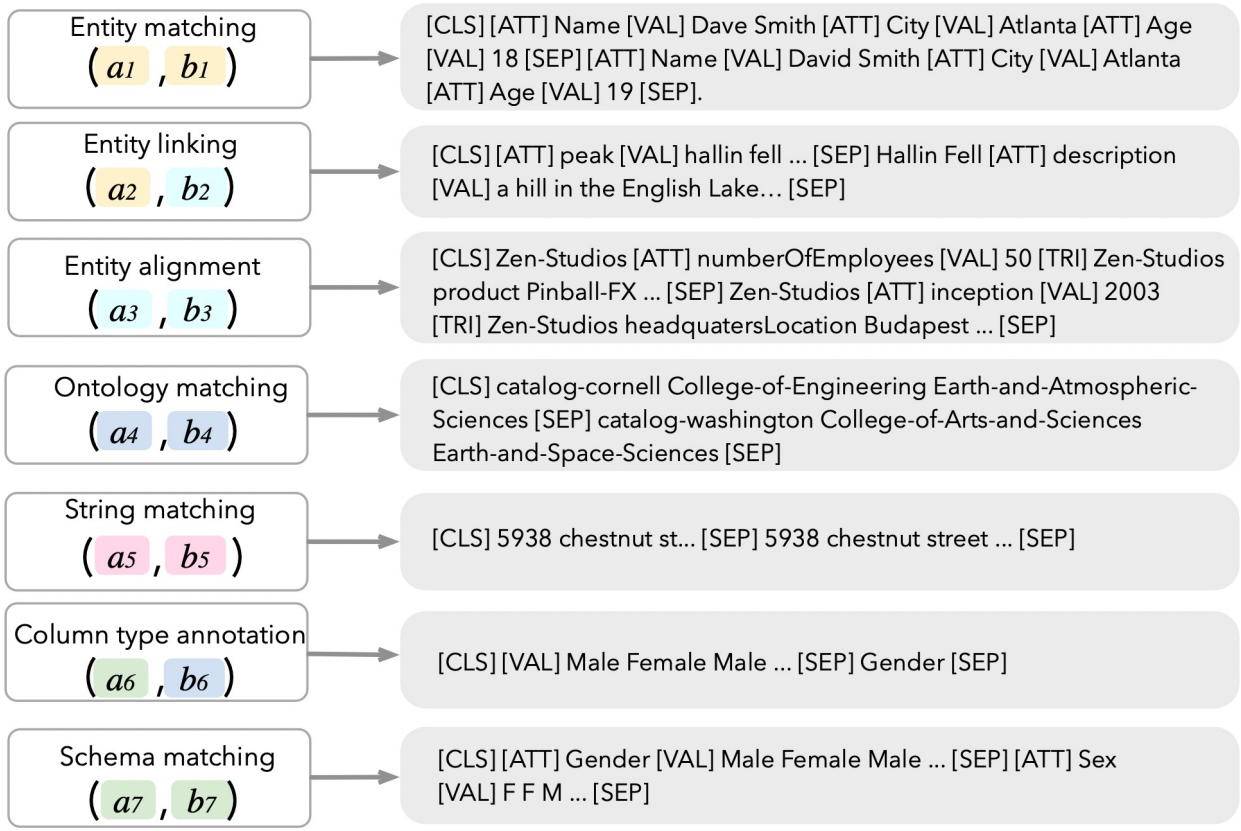
- **Pair-to-Text Serialization**: Serializing any pair of elements into text while still preserving their inherent structure
  - ➢Template

$$x = \mathrm{S}(a, b) = [\mathrm{CLS}]\ \mathrm{S}(a)[\mathrm{SEP}]\ \mathrm{S}(b)\ [\mathrm{SEP}]$$

  - ➢Instruction

$$x = \mathrm{S}(a, b) = [\mathrm{CLS}]\ \text{does}\ \mathrm{S}(a)\ [\mathrm{SEP}]\ \text{match with}\ \mathrm{S}(b)\ [\mathrm{SEP}]$$

- Employing a unified **Pre-trained Language Model** for effective encoding
  - ➢Which PLM is best for Unicorn?

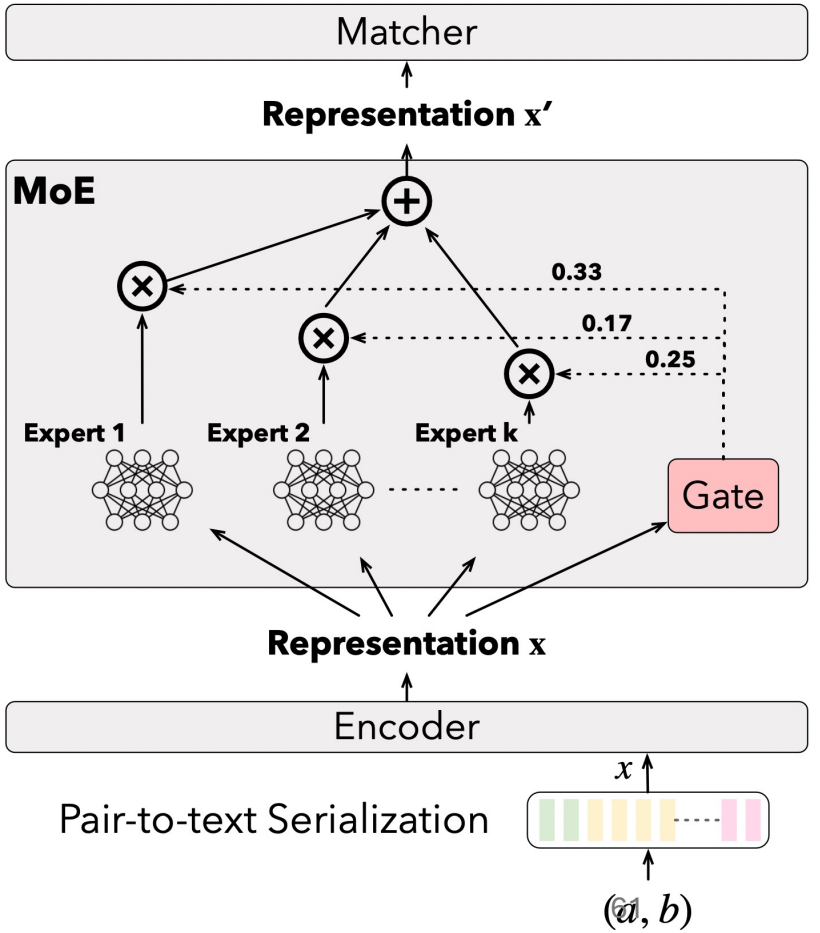


Mixture-of-Experts (MoE) Layer

- The objective of the MoE in Unicorn is to map different distributions of multiple tasks to a same shared distribution
- **Idea:** to divide the input feature space into sub-spaces
- **Experts** are a set of neural networks, whose parameters are not shared, to individually map representation x to a new representation x'
- **Gate** combines the outputs of Experts according to different weights

$$\mathbf{x}' = \mathrm{MoE}(\mathbf{x}) = g_1 \cdot \mathbf{x}_1 + g_2 \cdot \mathbf{x}_2 + \cdots + g_k \cdot \mathbf{x}_k$$

- This allows to share common knowledge
  - in **multi-task learning**
  - maintaining the characteristics of a single task



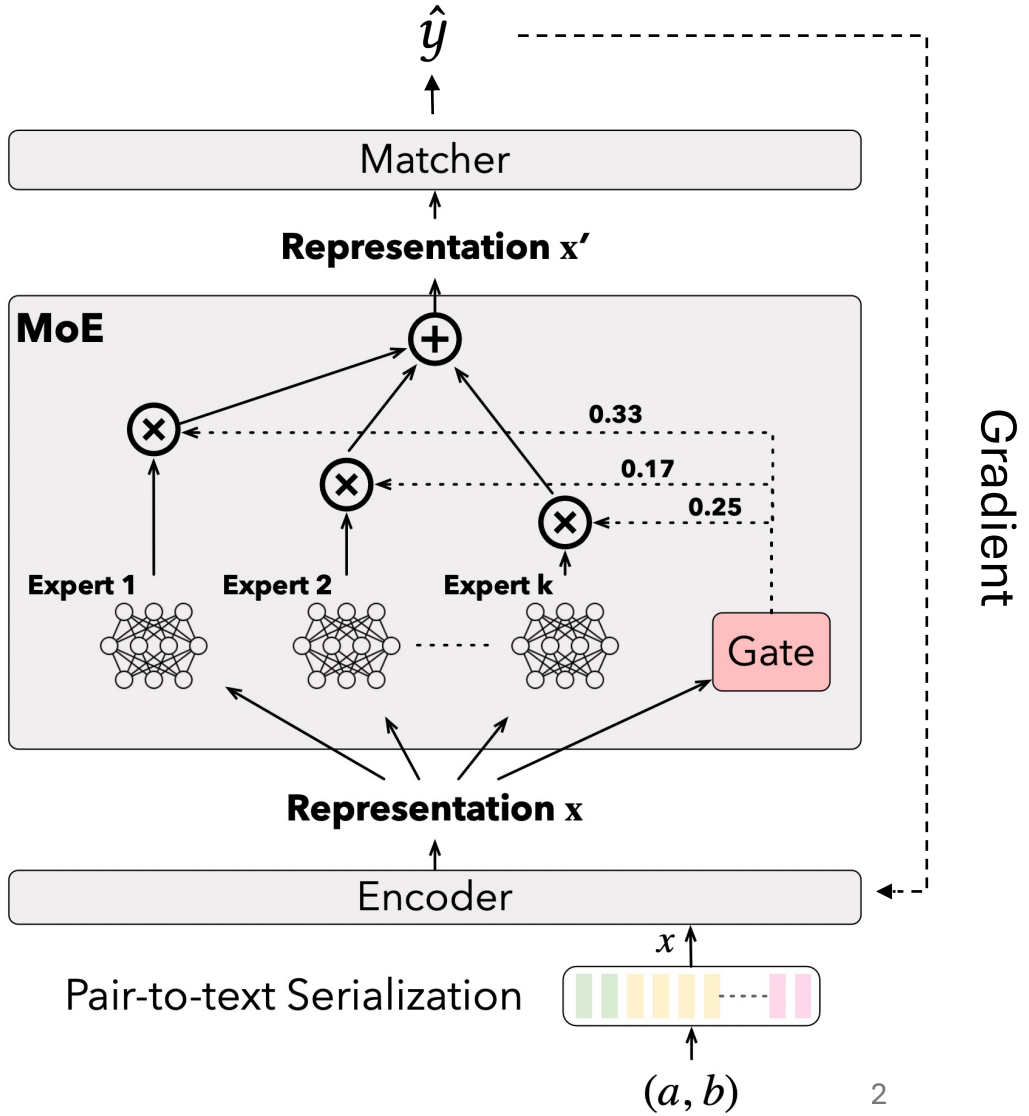
End-to-End Model Training

- Given all labeled pairs from all matching tasks, Unicorn predicts result:

$$\hat{y} = \mathrm{M}(\mathbf{x}')$$

- Computing the cross entropy loss and **optimizing parameters** of Unicorn:

$$\mathcal{L} = \mathbb{E}_{(\mathbf{x}', y) \in \mathcal{D}} \mathcal{L}_{\mathrm{CE}}(\hat{y}, y)$$



MoE Optimization for Expert Routing

- One obstacle of training MoE is that various input vectors {x} may use the same few Experts, which fails to take advantage of different Experts for various data matching tasks
- **Optimization strategy for Expert Routing**, i.e., 2 more objectives during training

$$\mathcal{L}_{\text{new}} = \mathcal{L} + \mathcal{L}_{\text{Bal}} + \mathcal{L}_{\text{Ent}}$$

① **Load balancing loss**: during training, all the experts have to be used in a balanced way

$$\mathbf{u} = (\sum_{\mathbf{x}\in\mathcal{D}} g_1, \sum_{\mathbf{x}\in\mathcal{D}} g_2, \ldots, \sum_{\mathbf{x}\in\mathcal{D}} g_k) \qquad \mathcal{L}_{\text{Bal}} = [\frac{\sigma(\mathbf{u})}{\mu(\mathbf{u})}]^2$$

② **Entropy loss**: for any specific training pair, we would like to assign a few specific experts to it, instead of evenly assigning it to all the Experts

$$\mathcal{L}_{\text{Ent}} = \mathbb{E}_{(\mathbf{x}',y)}\text{Entropy}(\mathbf{g}) = \mathbb{E}_{(\mathbf{x}',y)} - \sum_{i=1}^{k} g_i \cdot \log(g_i)$$

Experiment on 7 task types

Task Type	Task	$\|A\|$	$\|B\|$	# Matches	# Non-Matches	Metric
Entity Matching (EM) (Tuple, Tuple)	Walmart-Amazon (WA)	2,554	22,074	962	9,280	F1
	DBLP-Scholar (DS)	2,616	64,263	5,347	23,360	F1
	Fodors-Zagats (FZ)	533	331	110	836	F1
	iTunes-Amazon (IA)	6,907	55,923	132	407	F1
	Beer (Be)	4,345	3,000	68	382	F1
Column Type Annotation (CTA) (Column, Ontology)	Efthymiou (Ef)	620	31	620	18,600	Acc.
	T2D (T2D)	383	37	383	13,788	Acc.
	Limaye (Lim)	174	27	179	4,519	Acc.
Entity Linking (EL) (Tuple, KG-Entity)	T2D (T2D)	11,650	26,025	20,666	131,945	F1
	Limaye (Lim)	659	4,166	1,447	36,020	F1
String Matching (StM) (String, String)	Address (Ad)	24,650	29,531	9,850	1,062	F1
	Names (Na)	10,341	15,396	5,132	2,763	F1
	Researchers (Re)	8,342	43,549	4,556	4,767	F1
	Product (Pr)	2,554	22,074	1,154	79,310	F1
	Citation (Ci)	2,616	64,263	5,347	34,152	F1
Schema Matching (ScM) (Column, Column)	FabricatedDatasets (Fa)	11,172	11,352	7,692	109,762	Recall
	DeepMDatasets (DM)	41	41	41	268	Recall
Ontology Matching (OM) (Ontology, Ontology)	Cornell-Washington (CW)	176	166	53	285	Acc.
Entity Alignment (EA) (KG-Entity, KG-Entity)	SRPRS: DBP-YG (SYG)	15,000	15,000	15,000	38,891	Hits@K
	SRPRS: DBP-WD (SWD)	15,000	15,000	15,000	38,492	Hits@K

Evaluation on Encoder

Type	Task	Metric	BERT	RoBERTa	DistilBERT	DistilRoBERTa	XLNet	MPNet	DeBERTa
EM	Walmart-Amazon	F1	84.24	84.37	75.07	68.99	79.43	83.01	**86.89**
	DBLP-Scholar	F1	**95.7**	94.88	93.92	95.53	95.62	95.02	95.64
	Fodors-Zagats	F1	97.67	95.24	92.68	95.24	**100**	97.67	**100**
	iTunes-Amazon	F1	94.55	96.3	90	87.1	89.66	94.55	**96.43**
	Beer	F1	87.5	**90.32**	86.67	**90.32**	84.85	**90.32**	**90.32**
CTA	Efthymiou	Acc.	**98.43**	98.1	97.51	97.68	97.58	98.13	98.42
	T2D	Acc.	**99.29**	98.75	98.22	98.27	98.17	98.89	99.14
	Limaye	Acc.	96.63	96.35	96.26	96.2	96.2	96.63	**96.75**
EL	T2D	F1	**93.65**	89.3	85.08	78.86	78.85	92.17	91.96
	Limaye	F1	85.08	85.8	66.71	78.94	82.53	83.37	**86.78**
StM	Address	F1	98.49	98.59	96.75	97.13	98.15	98.56	**98.68**
	Names	F1	92.58	**94.37**	55.47	90.71	76.26	80.97	91.19
	Researchers	F1	**98.99**	97.71	97.88	95.65	98.72	98.44	97.66
	Product	F1	81.13	80.67	65.8	63.71	71.73	76.81	**82.9**
	Citation	F1	**96.28**	95.7	93.74	95.11	95.91	95.28	96.27
ScM	FabricatedDatasets	Recall	77.85	77.72	42.96	77.48	70.99	70.62	**89.6**
	DeepMDatasets	Recall	88.89	88.89	92.59	92.59	**100**	88.89	96.3
OM	Cornell-Washington	Acc.	90.64	86.38	91.06	68.94	77.45	75.74	**92.34**
EA	SRPRS: DBP-YG	Hits@1	99.34	99.49	99.24	99.22	98.69	99.47	**99.67**
	SRPRS: DBP-WD	Hits@1	97.3	97.13	97.34	96.62	96.78	**97.47**	97.22
AVG			92.71	92.3	85.75	88.21	89.38	90.6	**94.21**

rmers

Finetuning DeBERTa[1] achieves nearly the best results on almost all datasets of our seven data matching tasks

[1] P. He, X. Liu, J. Gao, W. Chen. Deberta: decoding-enhanced bert with disentangled attention. ICLR 2021,

Overall Results on Unified Prediction

Type	Task	Metric	Unicorn w/o MoE	Unicorn	Unicorn ++	Previous SOTA (Paper)
EM	Walmart-Amazon	F1	85.12	86.89	**86.93**	86.76 (Ditto [30])
	DBLP-Scholar	F1	95.38	95.64	**96.22**	95.6 (Ditto [30])
	Fodors-Zagats	F1	97.78	**100**	97.67	**100** (Ditto [30])
	iTunes-Amazon	F1	94.74	96.43	**98.18**	97.06 (Ditto [30])
	Beer	F1	90.32	90.32	87.5	**94.37** (Ditto [30])
CTA	Efthymiou	Acc.	98.08	98.42	**98.44**	90.4 (TURL [10])
	T2D	Acc.	98.81	99.14	**99.21**	96.6 (HNN+P2Vec [5])
	Limaye	Acc.	96.11	96.75	**97.32**	96.8 (HNN+P2Vec [5])
EL	T2D	F1	79.96	91.96	**92.25**	85 (Hybrid I [20])
	Limaye	F1	83.12	86.78	**87.9**	82 (Hybrid II [20])
StM	Address	F1	97.81	98.68	99.47	**99.91** (Falcon [39])
	Names	F1	86.12	91.19	**96.8**	95.72 (Falcon [39])
	Researchers	F1	96.59	97.66	**97.93**	97.81 (Falcon [39])
	Product	F1	84.61	82.9	**86.06**	67.18 (Falcon [39])
	Citation	F1	96.34	96.27	**96.64**	90.98 (Falcon [39])
ScM	FabricatedDatasets	Recall	81.19	**89.6**	89.35	81 (Valentine [27])
	DeepMDatasets	Recall	66.67	96.3	96.3	**100** (Valentine [27])
OM	Cornell-Washington	Acc.	90.64	**92.34**	90.21	80 (GLUE [15])
EA	SRPRS: DBP-YG	Hits@1	99.46	99.67	99.49	**100** (BERT-INT [46])
	SRPRS: DBP-WD	Hits@1	97.11	97.22	97.28	**99.6** (BERT-INT [46])
AVG			90.8	94.21	**94.56**	91.84
Model Size			139M	147M	147M	995.5M

- Unicorn is comparable or even outperforms specific models
- Unicorn++ > Unicorn > Unicorn w/o MoE

Better performance
Simpler structure
Smaller size