Probabilistic Management of OCR Data using an RDBMS

Arun Kumar
University of Wisconsin-Madison
arun@cs.wisc.edu

Christopher Ré
University of Wisconsin-Madison
chrisre@cs.wisc.edu

January 5, 2012

Abstract

The digitization of scanned forms and documents is changing the data sources that enterprises manage. To integrate these new data sources with enterprise data, the current state-of-the-art approach is to convert the images to ASCII text using optical character recognition (OCR) software and then to store the resulting ASCII text in a relational database. The OCR problem is challenging, and so the output of OCR often contains errors. In turn, queries on the output of OCR may fail to retrieve relevant answers. State-of-the-art OCR programs, e.g., the OCR powering Google Books, use a probabilistic model that captures many alternatives during the OCR process. Only when the results of OCR are stored in the database, do these approaches discard the uncertainty. In this work, we propose to retain the probabilistic models produced by OCR process in a relational database management system. A key technical challenge is that the probabilistic data produced by OCR software is very large (a single book blows up to 2GB from 400kB as ASCII). As a result, a baseline solution that integrates these models with an RDBMS is over 1000x slower versus standard text processing for single table select-project queries. However, many applications may have quality-performance needs that are in between these two extremes of ASCII and the complete model output by the OCR software. Thus, we propose a novel approximation scheme called STACCATO that allows a user to trade recall for query performance. Additionally, we provide a formal analysis of our scheme's properties, and describe how we integrate our scheme with standard-RDBMS text indexing.

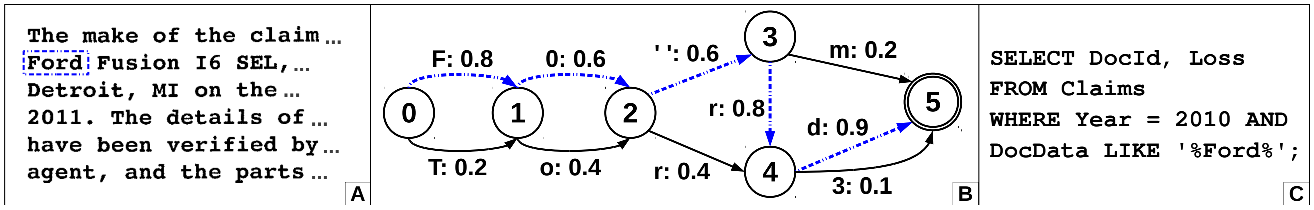
1 Introduction

The mass digitization of books, printed documents, and printed forms is changing the types of data that companies and academics manage. For example, Google Books and their academic partner, the Hathi Trust, have the goal of digitizing all of the world's books to allow scholars to search human knowledge from the pre-Web era. The hope of this effort is that digital access to this data will enable scholars to rapidly mine these vast stores of text for new discoveries.[1] The potential users of this new content are not limited to academics. The market for *enterprise document capture* (scanning of forms) is already in the multibillion dollar range [3]. In many of the applications, the translated data is related to enterprise business data, and so after converting to plain text, the data are stored in an RDBMS [6].

Translating an image of text (e.g., a jpeg) to ASCII is difficult for machines to do automatically. To cope with the huge number of variations in scanned documents, e.g., in spacing of the glyphs

[1]Many repositories of *Digging into Data Challenge* (a large joint effort to bring together social scientists with data analysis) are OCR-based `http://www.diggingintodata.org`.

and font faces, state-of-the-art approaches for optical character recognition (OCR) use probabilistic techniques. For example, the OCRopus tool from Google Books represents the output of the OCR process as a stochastic automaton called a *finite-state transducer* (FST) that defines a probability distribution over all possible strings that could be represented in the image.[2] An example image and its resulting (simplified) transducer are shown in Figure 1. Each labeled path through the transducer corresponds to a potential string (one multiplies the weights along the path to get the probability of the string). Only to produce the final plain text do current OCR approaches remove the uncertainty. Traditionally, they choose to retain only the single most likely string produced by the FST (called a *maximum a priori estimate* or MAP [1]).



```
The make of the claim…
Ford Fusion I6 SEL,…
Detroit, MI on the…
2011. The details of…
have been verified by…
agent, and the parts…
```

```
SELECT DocId, Loss
FROM Claims
WHERE Year = 2010 AND
DocData LIKE '%Ford%';
```

Figure 1: (A) An image of text. (B) A portion of a simple FST resulting from the OCR of the highlighted part of (A). The numbers on the arcs are conditional probabilities of transitioning from one state to another. An emitted string corresponds to a path from states 0 to 5. The string *'F0 rd'* (highlighted path) has the highest probability, $0.8*0.6*0.6*0.8*0.9 \approx 0.21$. (C) An SQL query to retrieve loss information that contains *'Ford'*. Using the MAP approach, no claim is found. Using STACCATO, a claim is found (with probability 0.12).

As Google Books demonstrates, the MAP works well for browsing applications. In such applications, one is sensitive to precision (i.e., are the answers I see correct), but one is insensitive to recall (i.e., what fraction of all of the answers in my corpus are returned). But this is not true of all applications: an English professor looking for the earliest dates that a word occurs in a corpus is sensitive to recall [5]. As is an insurance company that wants all insurance claims that were filled in 2010 that mentioned a *'Ford'*. This latter query is expressed in SQL in Figure 1(C). In this work, we focus on such single table select-project queries, whose outputs are standard probabilistic RDBMS tables. Using the MAP approach may miss valuable answers. In the example in Figure 1, the most likely string does not contain *'Ford'*, and so we (erroneously) miss this claim. However, the string *'Ford'* does appear (albeit with a lower probability). Empirically, we show that the recall for simple queries on real-world OCR can be as low as 0.3 – and so we may throw away almost 70% of our data if we follow the MAP approach.

To remedy this recall problem, our baseline approach is to store and handle the FSTs as binary large objects inside the RDBMS. As with a probabilistic relational database, the user can then pose questions as if the data are deterministic and it is the job of the system to compute the confidence in its answer. By combining existing open-source tools for transducer composition [3] with an RDBMS, we can then answer queries like that in Figure 1(C). This approach achieves a high quality (empirically, the recall we measured is very close to 1.0, with up to 0.9 precision). Additionally, the enterprise users can ask their existing queries directly on top of the RDBMS data (the query in Figure 1(C) remains unchanged). The downside is that query processing is much slower (up to 1000x slower). While the query processing time for transducers is linear in the data

[2]`http://code.google.com/p/ocropus/`.

[3]OpenFST. `http://www.openfst.org/`

size, the transducers themselves are huge, e.g., a single 200-page book blows up from 400 kB as text to over 2 GB when represented by transducers after OCR. This motivates our central question: *"Can we devise an approximation scheme that is somewhere in between these two extremes of recall and performance?"*

State-of-the-art OCR tools segment each of the images corresponding to pages in a document into lines using special purpose line-breaking tools. Breaking a single line further into individual words is more difficult (spacing is very difficult to accurately detect). With this in mind, a natural idea to improve the recall of the MAP approach is to retain not only the highest probability string for each line, but instead to retain the k highest probability strings that appear in each line (called k-MAP [30, 56]). Indeed, this technique keeps more information around at a linear cost (in k) in space and processing time. However, we show that even storing hundreds of paths makes an insignificant jump in the recall of queries.

To combat this problem, we propose a novel approximation scheme called STACCATO, which is our main technical contribution. The main idea is to apply k-MAP not to the whole line, but to first break the line into smaller chunks which are themselves transducers and apply k-MAP to each transducer individually. This allows us to store exponentially more alternatives than k-MAP (exponential in the number of chunks), while using roughly a linear amount more space than the MAP approach. If there is only a single chunk, then STACCATO's output is equivalent to k-MAP. If essentially every possible character is a chunk, then we retain the full FST. Experimentally, we demonstrate that the STACCATO approach *gracefully trades off between performance and recall.* For example, when looking for mentions of laws on a data set that contains scanned acts of the US congress, the MAP approach achieves a recall of 0.28 executing in about 1 second, the full FST approach achieves perfect recall but takes over 2 minutes. An intermediate representation from STACCATO takes around 10 seconds and achieves 0.76 recall. Of course, there is a fundamental trade off between precision and recall. On the same query as above, the MAP has precision 1.0, and the full FST has precision 0.25, while STACCATO achieves 0.73. In general, STACCATO's precision falls in between the MAP and the full FST.

To understand STACCATO's approximation more deeply, we conduct a formal analysis, which is our second technical contribution. When constructing STACCATO's approximation, we ensure two properties (1) each chunk forms a transducer (as opposed to a more general structure), and (2) that the model retains the *unique path property*, i.e., that every string corresponds to a unique path. While both of these properties are satisfied by the transducers produced by OCRopus, neither property is necessary to have a well-defined approximation scheme. Moreover, enforcing these two properties increases the complexity of our algorithm and may preclude some compact approximations. Thus, it is natural to wonder if we can relax these two properties. While we cannot prove that these two conditions are necessary, we show that without these two properties, basic operations become intractable. Without the unique path property, prior work has shown that determining (even approximating) the k-MAP is intractable for a fixed k [34]. Even with the unique path property and a fixed set of chunks, we show that essentially the simplest violation of property (1) makes it intractable to construct an approximation even for $k = 2$ (Theorem 3.1). On the positive side, for any fixed partition, STACCATO retains a set of strings that achieves the highest total probability among approximations that satisfy the above restrictions.

Finally, we describe how to use standard text-indexing techniques to improve query performance. Directly applying an inverted index to transducer data is essentially doomed to failure: the sheer number of terms one would have to index grows exponentially with the length of the document, e.g., an FST for a single line may represent over 10^{100} terms. To combat this, we allow the user to specify a dictionary of terms. We then construct an index of those terms specified in the dictionary. This allows us to process keyword and some regular expressions using standard techniques [14, 55].

Outline In Section 2, we illustrate our current prototype system to manage OCR data using an RDBMS with an example, and we present a brief background on the use of transducers in OCR. In Section 3, we briefly describe the baseline solutions, and then discuss the main novel technical contributions of this work, viz., the Staccato approximation scheme and our formal analysis of its properties. In Section 4, we describe our approach for indexing OCR transducer data, which is another technical contribution of this work. In Section 5, we empirically validate that our approach is able to trade off recall for query-runtime performance on several real-world OCR data sets. We validate that our approximation methods can be efficiently implemented, and that our indexing technique provides the expected speedups. In Section 6, we discuss related work.

2 Preliminaries

The key functionality that Staccato provides is to enable users to query OCR data inside an RDBMS as if it were regular text. Specifically, we want to enable the LIKE predicate of SQL on OCR data. We describe Staccato through an example, followed by a more detailed explanation of its semantics and the formal background.

2.1 Using Staccato with OCR

Consider an insurance company that stores loss data with scanned report forms in a table with the following schema:

$$\textbf{Claims}(DocID, Year, Loss, DocData)$$

A document tuple contains an id, the year the form was filed (Year), the amount of the loss (Loss) and the contents of the report (DocData). A simple query that an insurance company may want to ask over the table - *"Get loss amounts of all claims in 2010 where the report mentions 'Ford' "*. Were DocData ASCII text, this could be expressed as an SQL query as follows:

```
SELECT DocID, Loss FROM Claims
WHERE Year = 2010 AND DocData LIKE '%Ford%';
```

If DocData is standard text, the semantics of this query is straightforward: we examine each document filed in 2010, and check if it contains the string *'Ford'*. The challenge is that instead of a single document, in OCR applications DocData represents many different documents (each document is weighted by probability). In Staccato, we can express this as an SQL query that uses a simple pattern in the `LIKE` predicate (also in Figure 1(C)). The twist is that the underlying processing must take into account the probabilities from the OCR model.

Formally, Staccato allows a larger class of queries in the LIKE predicate that can be expressed as deterministic finite automata (DFAs). Staccato translates the syntax above in to a DFA using standard techniques [29]. As with probabilistic databases [13, 24, 32, 53] , Staccato computes the probability that the document matches the regular expression. Staccato does this using algorithms from prior work [34, 45]. The result is a probabilistic relation; after this, we can apply probabilistic relational database processing techniques [24, 43, 48]. In this work, we consider only single table select-project queries (joins are handled using the above mentioned techniques).

A critical challenge that Staccato must address is given a DFA find those documents that are relevant to the query expressed by the DFA. For a fixed query, the existing algorithms are roughly linear in the size of data that they must process. To improve the runtime of these algorithms, one strategy (that we take) is to reduce the size of the data that must be processed using approximations. The primary contribution of Staccato is the set of mechanisms that we describe in Section 3 to achieve the trade off of quality and performance by approximating the data. We formally study the properties of our algorithms and describe simple mechanisms to allow the user to set these parameters in Sec. 3.2.

One way to evaluate the query above in the deterministic setting is to scan the string in each report and check for a match. A better strategy may be to use an inverted index to fetch only those documents that contain *'Ford'*. In general, this strategy is possible for *anchored* regular expressions [21], which are regular expressions that begin or end with words in the language, e.g. 'no.(2|3)' is anchored while '(no|num).(2|8)' is not. Staccato supports a similar optimization using standard text-indexing techniques. There is, however, one twist: At one extreme, any term may have some small probability of occurring at every location of the document – which renders the index ineffective. Nevertheless, we show that Staccato is able to provide efficient indexing for anchored regular expressions using a dictionary-based approach.

2.2 Background: Stochastic Finite Automata

We formally describe Staccato's data model that is based on Stochastic Finite Automata (SFA). This model is essentially identical to the model output by Google's OCRopus [8, 41].[4] An SFA is a finite state machine that emits strings (e.g., the ASCII conversion of an OCR image). The model is stochastic, which captures the uncertainty in translating the glyphs and spaces to ASCII characters.

At a high level, an SFA over an alphabet Σ represents a discrete probability distribution P over strings in Σ^*, i.e.,

$$P : \Sigma^* \to [0,1] \text{ such that } \sum_{x \in \Sigma^*} P(x) = 1$$

The SFA represents the (finitely many) strings with non-zero probability using an automaton-like structure that we first describe using an example:

Example 1. Figure 1 shows an image of text and a simplified SFA created by OCRopus from that data. The SFA is a directed acyclic labeled graph. The graphical structure (i.e., the branching) in the SFA is used by the OCR tool to capture correlations between the emitted letters. Each source-to-sink path (i.e., a path from node 0 to node 5) corresponds to a string with non-zero probability. For example, the string *'Ford'* is one possible path that uses the following sequence of nodes $0 \to 1 \to 2 \to 4 \to 5$. The probability of this string can be found by multiplying the edge weights corresponding to the path: $0.8 * 0.4 * 0.4 * 0.9 \approx 0.12$. □

[4]Our prototype uses the same weighted finite state transducer (FST) model that is used by OpenFST and OCRopus. We simplify FST to SFAs here only slightly for presentation. See the full version for more details [36]

Formally, we fix an alphabet Σ (in STACCATO, this is the set of ASCII characters). An SFA S over Σ is a tuple $S = (V, E, s, f, \delta)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges such that (V, E) is a directed acyclic graph, and s (resp. f) is a distinguished start (resp. final) node. The function δ is a stochastic transition function, i.e.,

$$\delta : E \times \Sigma \to [0,1] \text{ s.t. } \sum_{\substack{y:(x,y)\in E \\ \sigma \in \Sigma}} \delta((x,y),\sigma) = 1 \quad \forall x \in V$$

In essence, $\delta(e, \sigma)$, where $e = (x, y)$, is the conditional probability of transitioning from $x \to y$ and emitting σ.

An SFA defines a probability distribution via its labeled paths. A labeled path from s to f is denoted by $p = (e_1, \sigma_1), \ldots, (e_N, \sigma_N)$, where $e_i \in E$ and $\sigma_i \in \Sigma$, corresponding to the string $\sigma_1 ... \sigma_n$, with its probability: [5]

$$\Pr_S[p] = \prod_{i=1}^{|p|} \delta(e_i, \sigma_i)$$

SFAs in OCR satisfy an important property that we call the *unique paths property* that says that any string produced by the SFA with non-zero probability is generated by a unique labeled path through the SFA. We denote by UP the function that takes a string to its unique labeled path. This property guarantees tractability of many important computations over SFAs including finding the highest probability string produced by the SFA [34].

Unlike the example given here, the SFAs produced by Google's OCRopus are much larger: they contain a weighted arc for every ASCII character. And so, the SFA for a single line can require as much as 600 kB to store.

Queries in STACCATO are formalized in the standard way for probabilistic databases. In this paper, we consider LIKE predicates that contain Boolean queries expressed as DFAs (STACCATO handles non-Boolean queries using algorithms in Kimmelfeld and Ré [34]). Fix an alphabet Σ (the ASCII characters). Let $q : \Sigma^* \to \{0, 1\}$ be expressed as DFA and x be any string. We have $q(x) = 1$ when x satisfies the query, i.e., it's accepted by the DFA. We compute the probability that q is true; this quantity is denoted $\Pr[q]$ and is defined by $\Pr[q] = \sum_{x \in \Sigma^*} q(x) \Pr(x)$ (i.e., simply sum over all possible strings where q is true). There is a straightforward algorithm based on matrix multiplication to process these queries that is linear in the size of the data and cubic in the number of states of the DFA [45].

3 Managing SFAs in an RDBMS

We start by outlining two baseline approaches that represent the two extremes of query performance and recall. Then, we describe the novel approximation scheme of STACCATO, which enables us to trade performance for recall.

Baseline Approaches We study two baseline approaches: k-MAP and the FullSFA approach. Fix some $k \geq 1$. In the k-MAP approach we store the k highest probability strings (simply, top k strings) generated by each SFA in our databases. We store one tuple per string along with the

[5]Many (including OpenFST) tools use a formalization with log-odds instead of probabilities. It has some intuitive property for graph concepts, e.g., the shortest path corresponds to the most likely string.

associated probability. Query processing is straightforward: we process each string using standard text-processing techniques, and then sum the probability of each string (since each string is a disjoint probabilistic event). In the FullSFA approach, we store the entire SFA as a BLOB inside the RDBMS. To answer a query, we retrieve the BLOB, deserialize it, and then use an open source C++ automata composition library to answer the query [11, 12] and compute all probabilities. Table 1 summarizes the time and space costs for a simple chain SFA (no branching). This table gives an engineer's intuition about the time and space complexity of the baseline approaches. The factor 16 accounts for the metadata – tuple ID, location in SFA, and probability value (the schema is described in the full version [36]). We also include our proposed approach, Staccato that depends on a parameter m (the number of chunks) that we describe below. From the table, we can read that query processing time for Staccato is essentially linear in m. Let l be the length of the document, since $m \in [1, l]$ query processing time in Staccato interpolates linearly from the k-MAP approach to the FullSFA approach.

	k-MAP	FullSFA	Staccato
Query	lqk	$lq\|\Sigma\| + q^3(l-1)$	$lqk + q^3(m-1)$
Space	$lk + 16k$	$l\|\Sigma\| + 16l\|\Sigma\|$	$lk + 16mk$

l	:	length of the SFA's strings
q	:	# states in the query DFA
k	:	# paths parameter in k-MAP, Staccato
m	:	# chunks in Staccato ($1 \leq m \leq l$)

Table 1: Space costs and query processing times for a simple chain SFA. The space indicates the number of bytes of storage required.

3.1 Approximating an SFA with Chunks

As mentioned before, the SFAs in OCR are much larger than our example, e.g. one OCR line from a scanned book yielded an SFA of size 600 kB. In turn, the 200-page book blows up to over 2 GB when represented by SFAs. Thus, to answer a query that spans many books in the FullSFA approach, we must read a huge amount of data. This can be a major bottleneck in query processing. To combat this we propose to approximate an SFA with a collection of smaller-sized SFAs (that we call *chunks*). Our goal is to create an approximation that allows us to gracefully tradeoff from the fast-but-low-recall MAP approach to the slow-but-high-recall FullSFA approach.

Recall that the k-MAP approach is a natural first approximation, wherein we simply store the top-k paths in each of the per-line SFAs. This approach can increase the recall at a linear cost in k. However, as we demonstrate experimentally, simply increasing k is insufficient to tradeoff between the two extremes. That is, even for huge values of k we do not achieve full recall.

Our idea to combat the slow increase of recall starts with the following intuition: the more strings from the SFA we store, the higher our recall will be. We observe that if we store the top k in each of m smaller SFAs (that we refer to as 'chunks'), we effectively store k^m distinct strings. Thus, increasing the value of k increases the number of strings polynomially. In contrast, increasing m, the number of smaller SFAs, increases the number of paths *exponentially*, as illustrated in Figure 2. This observation motivates the idea that to improve quality, we should divide the SFA further.

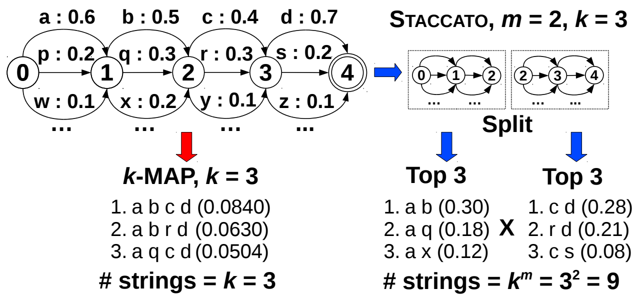


Figure 2: A depiction of conventional Top-k versus STACCATO's approximation.

Algorithm 1: FindMinSFA

```
**Inputs**: SFA $S$ with partial order $\leq$ on its nodes, $X \subseteq V$
**while** $X$ *does not form a valid SFA* **do**
    **if** *No unique start node in* $X$ **then**
        *Compute the least common ancestor of* $X$*, say,* $l$
        $X \leftarrow X \cup \{y \in V \mid l \leq y \text{ and } \forall x \in X, y \leq x\}$
    **if** *No unique end node in* $X$ **then**
        *Compute greatest common descendant of* $X$*, say,* $g$
        $X \leftarrow X \cup \{y \in V \mid y \leq g \text{ and } \forall x \in X, x \leq y\}$
    $\forall e \in E$ *s.t. exactly one end-point is in* $X - \{l, g\}$*, add other end-point to* $X$
```

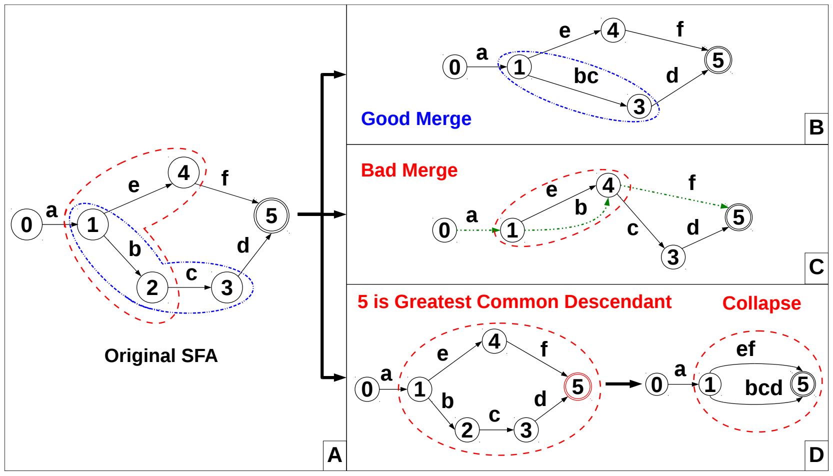


Figure 3: Algorithm 1: FindMinSFA. Illustrating merge and FindMinSFA: (A) Original: The SFA emits two strings: aef and $abcd$. Two merges considered: {(1,2),(2,3)} (successive edges), and {(1,2),(1,4)} (sibling edges). (B) Good merge: First set gives new edge (1,3), emitting bc. The SFA still emits only aef and $abcd$. (C) Bad merge: Second set gives new edge (1,4), emitting e and b. But, the SFA now wrongly emits new strings, e.g., abf (dashed lines). (D) Using Algorithm 1 on the second set, the greatest common descendant is obtained (node 5), and the resulting set is collapsed to edge (1,5). The SFA now emits only aef and $abcd$.

As we demonstrate experimentally, STACCATO achieves the most conceptually important feature of our approximation: it allows us to smoothly tradeoff recall for performance. In other words, increasing m (and k) increases the recall at the expense of performance.

SFA Approximation Given an SFA S, our goal is to find a new SFA S' that satisfies two competing properties: (1) S' should be smaller than S, and (2) the set of strings represented by S' should contain as many of the high probability strings from S as possible without containing any strings not in S.[6] Our technique to approximate the SFA S is to merge a set of transitions in S (a 'chunk') to produce a new SFA S'; then we retain only the top k transitions on each edge in S'.

To describe our algorithm, we need some notation. We generalize the definition of SFAs (Section 2) to allow transitions that produce strings (as opposed to single characters). Formally, the transition function δ has the type $\delta : E \times \Sigma^{+} \to [0, 1]$. Any SFA meets this generalized SFA definition, and so we assume this generalized definition of SFAs for the rest of the section.

Before describing the merging operation formally, we illustrate the challenge in the merging process in Figure 3. Figure 3(A) shows an SFA (without probabilities for readability). We consider two merging operations. First, we have chosen to merge the edges $(1, 2)$ and $(2, 3)$ and replaced it with a single edge $(1, 3)$. To retain the same strings that are present in the SFA in (A), the transition function must emit the string '*bc*' on the new edge $(1, 3)$ as illustrated in Figure 3(B). In contrast, if we choose to merge the edges $(1, 2)$ and $(1, 4)$, there is an issue: *no matter what we put on the transition from* $(1, 4)$ *we will introduce strings that are not present in the original SFA* (Figure 3(C)). The problem is that the set of nodes $\{1, 2, 4\}$ do not form an SFA by themselves (there is no unique final node). One could imagine generalizing the definition of SFA to allow richer structures that could capture the correlations between strings, but as we explain in Section 3.2, this approach creates serious technical challenges. Instead, we propose to fix this issue by searching for a minimal SFA S' that contains this set of nodes (the operation called FINDMINSFA). Then, we replace the nodes in the set with a single edge, retaining only the top k highest probability strings from S'. We refer to this operation of replacing S' with an edge as COLLAPSE. In our example, the result of these operations is illustrated in Figure 3(D).

We describe our algorithm's subroutine FINDMINSFA and then the entire heuristic.

FindMinSFA Given an SFA S and a set of nodes $X \subseteq V$, our goal is to find a SFA S' whose node set Y is such that that $X \subseteq Y$. We want the set Y to be minimal in the sense that removing any node $y \in Y$ causes S' to violate the SFA property, that is removing y causes S' to no longer have a unique start (resp. end) state. Presented in Algorithm 1, our algorithm is based on the observation that the unique start node s of S' must come before all nodes in X in the topological order of the graph (a partial order). Similarly, the end node f of the SFA S' must come after all nodes in X in topological order. To satisfy these properties, we repeatedly enlarge Y by computing the start (resp. final node) using the least common ancestor (resp. greatest common descendant) in the DAG. Additionally, we require that any edge in S that is incident to a node in Y can be incident to only either s or f. (Any node incident to both will be internal to S') If there are no such edges, we are done. Otherwise, for each such edge e, we include its endpoints in Y and repeat this algorithm with X enlarged to Y. Once we find a suitable set Y, we replace the set of nodes in the SFA S with a single edge from s (the start node of S') to f (the final node of S'). Figure 3(D)

[6]This is a type of *sufficient lineage approximation* [46].

illustrates a case when there is no unique end node, and the greatest common descendant has to be computed. More illustrations, covering the other cases, are presented in the full version [36].

Algorithm Description The inputs to our algorithm are the parameters k (the number of strings retained per edge) and m (the maximum number of edges that we are allowed to retain in the resulting graph). We describe how a user chooses these parameters in Section 3.2. For now, we focus on the algorithm. At each step, our approximation creates a restricted type of SFA where each edge emits at most k strings, i.e., $\forall e \in E$, $|\{\sigma \in \Sigma^* \mid \delta(e, \sigma) > 0\}| \leq k$. When given an SFA not satisfying this property, our algorithm chooses to retain those strings $\sigma \in \Sigma^*$ with the highest values of δ (ties broken arbitrarily). This set can be computed efficiently using the standard Viterbi algorithm [26], which is a dynamic programming algorithm for finding the most likely outputs in probabilistic sequence models, like HMMs. By memoizing the best partial results till a particular state, it can compute the globally optimal results in polynomial time. To compute the top-k results more efficiently, we use an incremental variant by Yen et al [54].

Algorithm 2: Greedy heuristic over SFA $S = (V, E)$

```
Choose $\{x, y, z\}$ s.t. $(x, y), (y, z) \in E$ and maximizing the probability mass of the retained strings.
$S \leftarrow$ Collapse(FindMinSFA($S, \{x, y, z\}$))
Repeat above steps till $|E| \leq m$
```

Algorithm 2 summarizes our heuristic: for each triple of nodes $\{x, y, z\}$ such that $(x, y), (y, z) \in E$, we find a minimal containing SFA S_{ij} by calling FindMinSFA($\{x, y, z\}$). We then replace the set of nodes in S_{ij} by a single edge f (Collapse above). This edge f keeps only the top-k strings produced by S_{ij}. Thus, the triple of nodes $\{x, y, z\}$ generates a candidate SFA. We choose the candidate such that the probability mass of all generated strings is as high as possible (note that since we have thrown away some strings, the total probability mass may be less than 1). Given an SFA we can compute this using the standard sum-product algorithm (a faster incremental variant is actually used in Staccato). We then continue to recurse until we have reached our goal of finding an SFA that contains fewer than m edges. A simple optimization (employed by Staccato) is to cache those candidates we have considered in previous iterations.

While our algorithm is not necessarily optimal, it serves as a proof of concept that our conceptual goal can be achieved. That is, Staccato offers a knob to tradeoff recall for performance. We describe the experimental setup in more detail in Section 5, but we illustrate our point with a simple experimental result. Figure 4 plots the recall and runtimes of the two baselines and Staccato. Here, we have set $k = 100$ and $m = 10$. On these two queries, Staccato falls in the middle on both recall and performance.

3.2 Extensions and Analysis

To understand the formal underpinning of our approach, we perform a theoretical analysis. Informally, the first question is: *"in what sense is choosing the k-MAP the best approximation for each chunk in our algorithm?"* The second question we ask is to justify our restriction to SFAs as opposed to richer graphical structures in our approximation. We show that k-MAP in each chunk

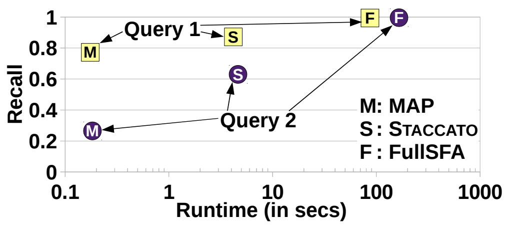


Figure 4: Recall - Runtime tradeoff for a keyword query (Query 1) and a regular expression query (Query 2). The parameters are: number of chunks $(m) = 10$, number of paths per chunk $(k) = 100$, and number of answers queried for $(NumAns) = 100$.

is no longer the best approximation and that there is likely no simple algorithm (as an underlying problem is NP-complete.)

We formally define the goal of our algorithms. Recall that an SFA S on Σ represents a probability distribution $\Pr_S : \Sigma^* \to [0,1]$. Given a set $X \subseteq \Sigma^*$, define $\Pr_S[X] = \sum_{x \in X} \Pr_s[x]$. All the approximations that we consider emit a subset of strings from the original model. Given an approximation scheme α, we denote by $\text{Emit}(\alpha)$ the set of strings that are emitted (retained) by that scheme. All other things being equal, we prefer a scheme α to α' whenever

$$\Pr_S[\text{Emit}(\alpha)] \geq \Pr_S[\text{Emit}(\alpha')]$$

That is, α retains more probability mass than α'. The formal basis for this choice is a standard statistical measure called the *Kullback-Leibler Divergence* [15], between the original and the approximate probability distributions. In the full version [36], we show that this divergence is lower (which means the approximate distribution is more similar to the original distribution) if the approximation satisfies the above inequality. In other words, a better approximation retains more of the high-probability strings.

We now describe our two main theoretical results. First for SFAs, STACCATO's approach to choosing the k highest probability strings in each chunk is optimal. For richer structures than SFAs, finding the optimal approximation is intractable (even if we are given the chunk structure, described below). Showing the first statement is straightforward, while the result about richer structures is more challenging.

Optimality of k-MAP for SFAs Given a generalized SFA $S = (V, \delta)$. Fix $k \geq 1$. Let $\mathbf{S}_{[k]}$ be the set of all SFAs (V, δ') that arise from picking k strings on each edge of S to store. That is, for any pair of nodes $x, y \in V$ the set of strings with non-zero probability has size smaller than k:

$$\left|\{\sigma \in \Sigma^* \mid \delta'((x,y),\sigma) > 0\}\right| \leq k$$

Let S_k denote an SFA that for each pair $(x,y) \in V$ chooses the highest probability strings in the model (breaking ties arbitrarily). Then,

Proposition 3.1. *For any $S' \in \mathbf{S}_{[k]}$, we have:*

$$\Pr_S[\text{Emit}(S_k)] \geq \Pr_S[\text{Emit}(S')]$$

Since S_k is selected by STACCATO, we view this as formal justification for STACCATO's choice.

Richer Structural Approximation We now ask a follow-up question: *"If we allow more general partitions (rather than collapsing edges), is k-MAP still optimal?"* To make this precise, we consider a partition of the underlying edges of the SFA into connected components (call that partition Φ). Keeping with our early terminology, an element of the partition is called a *chunk*. In each chunk, we select at most k strings (corresponding to labeled paths through the chunk). Let $\alpha : \Phi \times \Sigma^* \to \{0, 1\}$ be an indicator function such that $\alpha(\phi, \sigma) = 1$ only if in chunk ϕ we choose string σ. For any $k \geq 1$, let A_k denote the set of all such αs that picks at most k strings from each chunk, i.e., for any $\phi \in \Phi$ we have $|\{\sigma \in \Sigma^* \mid \alpha(\phi, \sigma) > 0\}| \leq k$. Let $\text{Emit}(\alpha)$ be the set of strings emitted by this representation with non-zero probability (all strings that can be created from concatenating paths in the model).

Following the intuition from the SFA case described above, the best α would select the k-highest probability strings in each chunk. However, this is not the case. Moreover, we exhibit chunk structures, where finding the optimal choice of α is NP-hard in the size of the structure. This makes it unlikely that there is any simple description of the optimal approximation.

Theorem 3.1. *Fix $k \geq 2$. The following problem is NP-complete. Given as input (S, Φ, λ) where S is an SFA, Φ partitions the underlying graph of S, and $\lambda \geq 0$, determine if there exists an $\alpha \in A_k$ satisfying* $\Pr[\text{Emit}(\alpha)] \geq \lambda$.

The above problem remains NP-complete if S is restricted to satisfy the unique path property and restricted to a binary alphabet. A direct consequence of this theorem is that finding the maximizer is at least NP-hard. We provide the proof of this theorem in the full version [36]. The proof includes a detailed outline of a reduction from a matrix multiplication-related problem that is known be to hard. The reduction is by a gadget construction that encodes matrix multiplication as SFAs. Each chunk has at most 2 nodes in either border (as opposed to an SFA which has a single start and final node). This is about the weakest violation of the SFA property that we can imagine, and suggests to us that the SFA property is critical for tractable approximations.

Automated Construction of Staccato Part of our goal is to allow knobs to trade recall for performance on a per application basis, but setting the correct values for m and k may be unintuitive for users. To reduce the burden on the user, we devise a simple parameter tuning heuristic that maximizes query performance, while achieving acceptable recall. To measure recall, the user provides a set of labeled examples and representative queries. The user specifies a quality constraint (average recall for the set of queries) and a size constraint (storage space as percentage of the original dataset size). The goal is to find a pair of parameters (m,k) that satisfies both these constraints. We note that the size of the data is a function of (m,k) (see Table 1), which along with the size constraint helps us express k in terms of m (or vice versa). We empirically observed that for a fixed size, a smaller m usually yields faster query performance than a smaller k, which suggests that we need to minimize the value of m to maximize query performance. Our method works as follows: we pick a given value of m, then calculate the corresponding k that lies on the size constraint boundary. Given the resulting (m,k) pair, we compute the STACCATO approximation of the dataset and estimate the average recall. This problem is now a one-dimensional search problem: our goal is to find the smallest m that satisfies the recall constraint. We solve this using essentially a binary search. If infeasible, the user relaxes one of the constraints and repeats the above method. We experimentally validated this tuning method and compared it with an exhaustive search on the parameter space. The results are discussed in Section 5.5.

4 Inverted Indexing

To speedup keywords and anchored regex queries on standard ASCII text, a popular technique is to use standard inverted-indexing [14]. While indexing k-MAP data is pretty straightforward, the FullSFA is difficult. The reason is that the FullSFA encodes exponentially many strings in its length, and so indexing all strings for even a moderate-sized SFA is hopeless. Figure 5 shows the size of the index obtained (in number of *postings* [14], in our case line number item pairs) when we try to directly index the Staccato text of a single SFA (one OCR line).

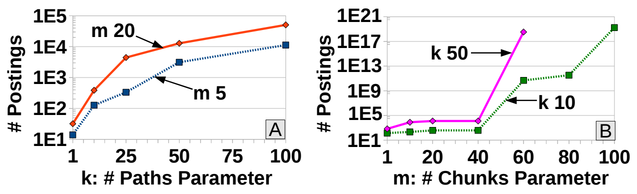


Figure 5: Number of postings (in logscale) from directly indexing one SFA. (A) Fix m, vary k. (B) Fix k, vary m. In (B), for $k = 50$, the number of postings overflows the 64-bit representation beyond $m = 60$.

Figure 5 shows an exponential blowup with m – which is not surprising as we store exponentially more paths with increasing m. Our observation is that many of these exponentially many terms are useless to applications. Thus, to extend the reach of indexing, we apply a standard technique. We use a dictionary of terms input by the user, and construct the index only for these terms [22]. These terms may be extracted from a known clean text corpus or from other sources like an English dictionary. Our construction algorithm builds a DFA from the dictionary of terms, and runs a slight modification of the SFA composition algorithm [29] with the data to find the start locations of all terms (details of the modification are in the full version [36]). The running time of the algorithm is linear in the size of the dictionary.

Projection In traditional text processing, given the length of the keyword and the offset of a match, we can read only that small portion of the document to process the query. We extend this idea to Staccato by finding a small portion of the SFA that is needed to answer the query – an operation that we call *projection*. Given a term t of length u, we obtain start locations of t from the postings. For each start location, we compute an (over)estimate of the nodes that we must process to obtain the term t. More precisely, we want the descendant nodes in the DAG that can be reached by a directed path from the start location that contains u or fewer edges (we find such nodes using a breadth-first search). This gives us a set of nodes that we must retrieve, which is often much smaller than the entire SFA.

We empirically show that even a simple indexing scheme as above can be used by Staccato to speedup keyword and anchored regular expression queries by over an order of magnitude versus a filescan-based approach. This validates our claim that indexing is possible for OCR transducers, and opens the possibility of adapting more advanced indexing techniques to improve the runtime speedups.

5 Experimental Evaluation

We experimentally verify that the Staccato approach can gracefully tradeoff between performance and quality. We also validate that our modifications to standard inverted indexing allow us to speedup query answering.

Dataset	No. of Pages	No. of SFAs	Size as: SFAs	Size as: Text
Cong. Acts (CA)	38	1590	533MB	90kB
English Lit. (LT)	32	1211	524MB	78kB
DB Papers (DB)	16	627	359MB	54kB

Table 2: Dataset Statistics. Each SFA represents one line of a scanned page.

Datasets Used We use three real-world datasets from domains where document digitization is growing. Congress Acts (CA) is a set of scans of acts of the U.S. Congress, obtained from The Hathi Trust [9]. English Literature (LT) is a set of scans of an English literature book, obtained from the JSTOR Archive [10]. Database Papers (DB) is a set of papers that we scanned ourselves to simulate a setting where an organization would scan documents for in-house usage. All the scan images were converted to SFAs using the OCRopus tool [8]. Each line of each document is represented by one SFA. We created a manual ground truth for these documents. The relevant statistics of these datasets are shown in Table 2. In order to study the scalability of the approaches on much larger datasets, we used a 100 GB dataset obtained from Google Books [7].

Experimental Setup The three approaches were implemented in C++ using PostgreSQL 9.0.3. The current implementation is single threaded so as to assess the impact of the approximation. All experiments are run on Intel Core-2 E6600 machines with 2.4 GHz CPU, 4 GB RAM, running Linux 2.6.18-194. The runtimes are averaged over 7 runs. The notation for the parameters is summarized in Table 3.

Symbol	Description
k	# Paths Parameter (k-MAP, Staccato)
m	# Chunks Parameter (Staccato)
$NumAns$	# Answers queried for

Table 3: Notations for Parameters

We set $NumAns = 100$, which is greater than the number of answers in the ground truth for all reported queries. If Staccato finds fewer matches than $NumAns$, it may return fewer answers. $NumAns$ affects precision, and we do sensitivity analysis for $NumAns$ in the full version [36].

5.1 Quality - Performance Tradeoff (Filescan)

We now present the detailed quality and performance results for queries run with a full filescan. The central technical claim of this paper is that Staccato bridges the gap from the low-recall-but-fast MAP to the high-recall-but-slow FullSFA. To verify this claim, we measured the recall and performance of 21 queries on the three datasets. We formulated these queries based on our discussions with practitioners in companies and researchers in the social sciences who work with real-world OCR data. Table 4 presents a subset of these results (the rest are presented in the full version of this paper [36]).

Query	MAP	k-MAP	FullSFA	STACCATO
	Precision/Recall			
CA1	1.00/0.79	1.00/0.79	0.14/1.00	1.00/0.79
CA2	1.00/0.28	1.00/0.52	0.25/1.00	0.73/0.76
LT1	0.96/0.87	0.96/0.90	0.92/1.00	0.97/0.91
LT2	0.78/0.66	0.76/0.66	0.31/0.97	0.44/0.81
DB1	0.93/0.75	0.90/0.92	0.67/0.99	0.90/0.96
DB2	0.96/0.76	0.96/0.76	0.33/1.00	0.91/0.97
	Runtime (in seconds)			
CA1	0.17	0.75	86.72	2.87
CA2	0.18	0.84	150.35	3.36
LT1	0.13	0.19	83.78	1.98
LT2	0.14	0.24	155.45	2.88
DB1	0.07	0.29	40.73	0.75
DB2	0.07	0.33	619.31	0.86

Table 4: Recall and runtime results across datasets. The keyword queries are – CA1: '*President*', LT1: '*Brinkmann*' and DB1: '*Trio*'. The regex queries are – CA2: '*U.S.C.* $2\backslash d\backslash d\backslash d$', LT2: '$19\backslash d\backslash d$, $\backslash d\backslash d$' and DB2: '$Sec(\backslash x)*\backslash d$'. Here, $\backslash x$ is any character and $\backslash d$ is any digit. The number of ground truth matches are – CA1: 28, LT1: 92, DB1: 68, CA2: 55, LT2: 32 and DB2: 33. The parameter setting here is: $k = 25,\ m = 40, NumAns = 100$.

We classify the kinds of queries to *keywords* and *regular expressions*. The intuition is that keyword queries are likely to achieve higher recall on k-MAP compared to more complex queries that contain spaces, special characters, and wildcards. Table 4 presents the recall and runtime results for six queries – one keyword and one regular expression (regex) query per dataset. Table 4 confirms that indeed there are intermediate points in our approximation that have faster runtimes than FullSFA (even up to two orders of magnitude), while providing higher quality than k-MAP.

We would like the tradeoff of quality for performance to be smooth as we vary m and k. To validate that our approximation can support this, we present two queries, a keyword and a regex, on the Congress Acts dataset (described below). To demonstrate this point, we vary k (the number of paths) for several values of m (the number of chunks) and plot the results in Figure 6. Given an SFA, m takes values from 1 to the number of the edges in the SFA (the latter being the nominal parameter setting 'Max'). When $m = 1$, STACCATO is equivalent to k-MAP. Note that the state-of-the-art in our comparison is essentially the MAP approach (k-MAP with $k = 1$, or STACCATO with $m = 1, k = 1$), which is what is employed by Google Books.

Keyword Queries In Figures 6 (A1) and (A2), we see the recall and performance behavior of running a keyword query (here *'President'*) in STACCATO for various combinations of k and m. We observe that the recall of k-MAP is high (0.8) but not perfect and in (A2) k-MAP is efficient (0.1s) to answer the query. Further, as we increase k there is essentially no change in recall (the

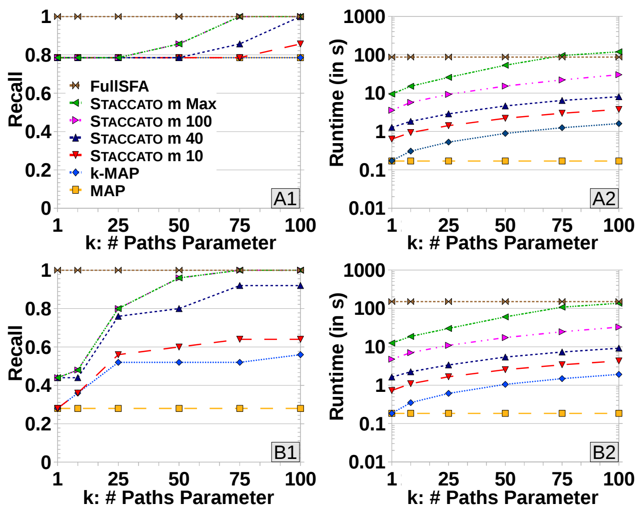


Figure 6: Recall and Runtime variations with k, for different values of m, on two queries: (A) '*President*' (keyword), and (B) '*U.S.C.* $2\backslash d\backslash d\backslash d$' (regex). The $\backslash d$ is short for (0|1|...|9). The runtimes are in logscale. $NumAns$ is set to 100. Recall that m is the number of chunks parameter and $NumAns$ is the number of answers queried for.

running time does increase by an order of magnitude). We verified that the reason is that the top-k paths change in only a small set of locations – and so no new occurrences of the string *'President'* are found. In contrast, the FullSFA approach achieves perfect recall, but it takes over 3 orders of magnitude longer to process the query. As we can see from the plots, for the STACCATO approach, the recall improves as we increase m – with corresponding slowdowns in query time. We believe that our approach is promising because of the gradual tradeoff of running time for quality. The fact that the k-MAP recall does not increase substantially with k, and does not manage to achieve the recall of FullSFA even for large k underscores the need for finer-grained partition, which is what STACCATO does.

Regular Expressions Figures 6 (B1) and (B2) present the results for a more sophisticated regex query that looks for a congressional code ('*U.S.C.* $2\backslash d\backslash d\backslash d$') referenced in the text. As the figure shows, this more sophisticated query has much lower recall for the MAP approach, and increases slowly with increasing k. Again, we see the same tradeoff that the FullSFA approach is orders of magnitude slower than k-MAP, but achieves perfect recall. Here, we see that the STACCATO approach does well: there are substantial (but smooth) jumps in quality as we increase k and m, going all the way from MAP to FullSFA. This suggests that more sophisticated queries benefit from our scheme more, which is an encouraging first step to enable applications to do rich analytics over such data.

Query Efficiency To assess the impact of query length on recall and runtime, we plot the two for a set of keyword queries of increasing length in Figure 7. We observe that the runtimes increase polynomially but slowly for all the approaches, while no clear trends exist for the recall. We saw similar results with regular expression queries, and discuss the details in the full version [36].

We also studied the impact of m and k on precision (and F-1 score), and observed that the

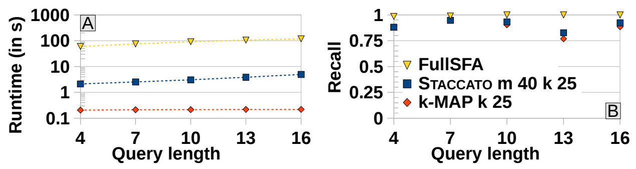


Figure 7: Impact of Query Length on (A) Runtime and (B) Recall. $NumAns$, the number of answers queried for, is set to 100.

precision of Staccato usually falls in between k-MAP and FullSFA (but F-1 of Staccato can be better than both in some cases). Similar to the recall-runtime tradeoff, Staccato also manages to gracefully tradeoff on precision and recall. Due to space constraints, these results are discussed in the full version [36].

5.2 Staccato Construction Time

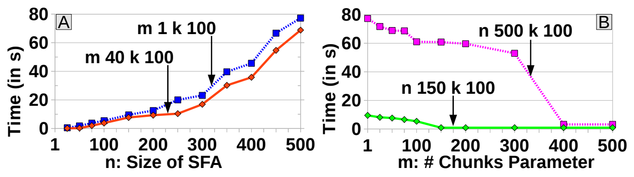


Figure 8: (A) Variation of Staccato approximation runtimes with the size of the SFA (n = number of nodes + edges) fixing m and k. (B) Sensitivity of the runtimes to m, fixing n and k. Recall that m is the number of chunks parameter and k is the number of paths parameter.

We now investigate the runtime of the Staccato's approximation algorithm. The runtime depends on the size of the input SFA data as well as m and k. We first fix m and k, then we plot the construction time for SFAs of varying size (number of nodes) from the CA dataset (Figure 8(A)). Overall, we can see that the algorithm runs efficiently – even in our unoptimized implementation. As this is an offline process, speed may not be critical for some applications. Also, this computation is embarassingly parallel (across SFAs). We used Condor [2] to run the Staccato construction on all the SFAs in the three datasets, for all of the above parameters. This process completed in approximately 11 hours.

To study the sensitivity of the construction time to m, we select a fixed SFA from the CA dataset (Figure 8(B)). When $m \geq |E|$, the algorithm picks each transition as a block, and terminates. But when $m = 300 < |E|$, the algorithm computes several candidate merges, leading to a sudden spike in the runtime. There onwards, the runtime varies almost linearly with decreasing m. However, there are some spikes in the middle. We verified that the spikes arise since the 'FindMinSFA' operation has to fix merged chunks not satisfying the SFA property, thus causing the variation to be less smooth. We also verified that the runtime was linear in k, fixing the SFA and m (see full version [36]). In general, a linear runtime in k is not guaranteed since the chunk structure obtained during merging may not be similar across k, for a given SFA and m.

5.3 Inverted Indexing

We now verify that standard inverted indexing can be made to work on SFAs. We implement the index as a relational table with a B+-tree on top of it. More efficient inverted indexing implementations are possible, and so our results are an upperbound on indexing performance. However, this prototype serves to illustrate our main technical point that indexing is possible for such data.

A dictionary of about 60,000 terms from a freely available dictionary [4] was converted to a prefix-trie automaton, and used for index construction. While parsing the query, we ascertain if the given regex contains a left-anchor term. If so, we look up the anchor in the index to obtain the postings, and retrieve the data to employ query processing on them.

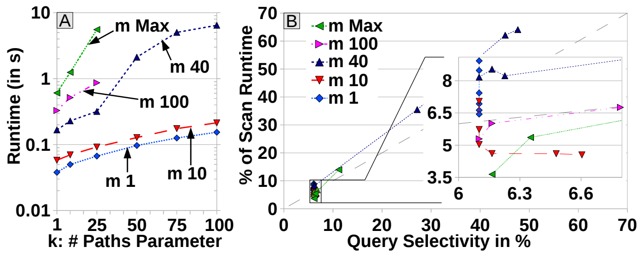


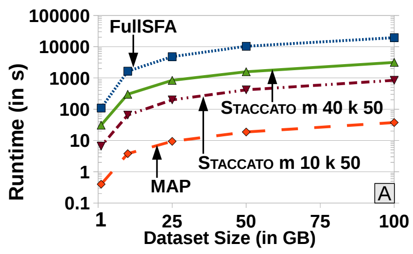
Figure 9: (A) Total Runtimes, and (B) Fractional Runtimes Vs Selectivity for the query '*Public Law* (8|9)\d', using the inverted index with the left anchor term '*public*'. Runtimes are in logscale. Recall that m is the number of chunks parameter.

Figure 9 shows the results for a fixed length left anchored regex on the CA data set that is anchored by a word in the dictionary (here, 'Public'). We omit some combinations ($m = 100, \text{Max}$ and $k = 50, 75, 100$) since their indexes had nearly 100% selectivity for all queries that we consider, rendering them useless. The first plot shows the sensitivity of the total runtimes to m and k. Mostly, there is a linear trend with k, except for a spike at $m = 40, k = 50$. To understand this behavior, we plot the runtime, as a percentage of the filescan runtime, against the selectivity of the term in the index. Ideally, the points should lie on the $Y = X$ line, or slightly above it. For the lowest values of m and k, the relative speedup is slightly lowered by the index lookup overhead. But as k increases, the query processing dominates, and hence the speedup improves, though selectivity changes only slightly. For higher m, the projection overhead lowers the speedup, and as k goes up, the selectivity shoots up, increasing the runtime. Overall, we see that dictionary-based indexing provides substantial speedups in many cases.

5.4 Scalability

To understand the feasibility of our approaches on larger amounts of data, we now study how the runtimes scale with increasing dataset sizes. We use a set of 8 scanned books from Google Books [7] and use OCRopus to obtain the SFAs. The total size of the SFA dataset is 100 GB.

Figure 10 shows the scalability results for a regex query. The filescans for FullSFA, MAP and Staccato all scale linearly in the dataset size. Overall, the filescan runtimes are in the order a few hours for FullSFA. The runtimes are one to two orders of magnitude lower for Staccato, depending on the parameters, and about three orders of magnitude lower for MAP. We also verified that indexing over this data provides further speedup (subject to query selectivity) as shown before.



Size (GB)	# SFAs
1	3400
10	32000
25	86000
50	172000
100	345000

B

Figure 10: (A) Filescan runtimes (logscale) against the dataset size for MAP, FullSFA and STACCATO with two parameter settings. (B) Number of SFAs in the respective datasets.

One can speedup query answering in all of the approaches by partitioning the dataset across multiple machines (or even using multiple disks). Thus, to scale to much larger corpora (say, millions of books), we plan to investigate the use of parallel data processing frameworks to attack this problem.

5.5 Automated Parameter Tuning

We now empirically demonstrate the parameter tuning method on a labeled set of 1590 SFAs (from the CA dataset), and a set of 5 queries (both keywords and regular expressions). The size constraint is chosen as 10% and the recall constraint is chosen as 0.9. We use increments of 5 for both m and k. Based on the tuning method described in Section 3.2, we obtain the following size equation: $20mk + 58k = 45540$, and the resultant parameter estimates of $m = 45$, $k = 45$, with a recall of 0.91. We then performed an exhaustive search on the parameter space to obtain the optimal values subject to the same constraints. Figure 11 shows the surface plots of the size and the recall obtained by varying m and k. The optimal values obtained are: $m = 35$, $k = 80$, again with a recall of 0.91. The difference in the parameter values arises primarily because the tuning method overestimated the size at this location. Nevertheless, we see that the tuning method provides parameter estimates satisfying the user requirements.

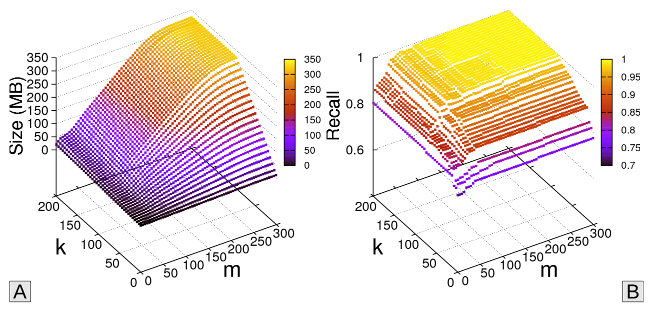


Figure 11: 3-D plots showing the variation of (A) the size of the approximated dataset (in MB), and (B) the average recall obtained. Recall that m is the number of chunks parameter and k is the number of paths parameter.

6 Related Work

Transducers are widely used in the OCR and speech communities [11, 41] and mature open-source tools exist to process in-memory transducers [12]. For example we use a popular open-source tool, OCRopus [8], from Google Books that provides well-trained language models and outputs transducers. See Mohri et al. [41] for a discussion of why transducers are well-suited to represent the uncertainty for OCR. In the same work, Mohri et al. also describe speech data. We experimented with speech data, but we were hampered by the lack of high quality open-source speech recognizer toolkits. Using the available toolkits, we found that the language quality from open source speech recognizers is substantially below commercial quality.

The Lahar system [39, 45] manages Hidden Markov Models (HMMs) as *Markovian streams* inside an RDBMS and allows querying them with SQL-like semantics. In contrast to an HMM [44] that requires that all strings be of the same length, transducers are able to encode strings of different lengths. This is useful in OCR, since identifying spaces between words is difficult, and this uncertainty is captured by the branching in the SFA [41]. Our work drew inspiration from the empirical study of work of approximation trade-offs from Letchner et al. [39]. Directly relevant to this work is the recent theoretical results of Kimelfeld and Ré [34], who studied the problem of evaluating transducers as queries over uncertain sequence data modeled using Hidden Markov Models [44, 45]. Staccato represents both the data and query by transducers which simplifies the engineering of our system.

Transducers are a graphical representation of probability models which makes them related to graphical models. Graphical models have been a hot topic in the database research community. Kanagal et al. [32] handle general graphical models. Wang et al. [52] also process Conditional Random Fields (CRFs) [37]. Though transducers can be viewed as a specialized directed graphical model, the primary focus of our work here is on the application of transducers to OCR in the domain of content management and the approximations that are critical to achieve good performance. However, our work is similar in spirit to these in that we too want to enable SQL-like querying of probabilistic OCR data inside an RDBMS.

Probabilistic graphical models have been successfully applied to various kinds of sequential data including OCR [19], RFID [45], speech [40], etc. Various models have been studied in both the machine learning and data management communities [23, 31, 32, 45, 52].

Many approximation schemes for probabilistic models have been studied [30, 39]. We built on the technique k-MAP [1], which is particularly relevant to us. Essentially, the idea is to infer the top k most likely results from the model and keep only those around. Another popular type of approximation is based on *mean-field theory*, where the intuition is that we replace complex dependencies (say in a graphical model) with their average (in some sense) [51]. Both mean-field theory and our approach share a common formal framework: minimizing KL-divergence. For a good overview of various probabilistic graphical models, approximation and inference techniques, we refer the reader to the excellent book by Wainwright and Jordan [51].

Gupta and Sarawagi [27] devise efficient approximation schemes to represent the outputs of a CRF, viz., labeled segmentations of text, in a probabilistic database. They partition the space of segmentations (i.e., the outputs) using boolean constraints on the output segment labels, and then structurally merge the partitions to a pre-defined count using Expectation Maximization, without any enumeration. Thus, their final partitions are disjoint sets of full-row outputs ('horizontally' partitioned). Both their approach and Staccato use KL-divergence to measure the goodness of approximation. However, Staccato is different in that we partition the underlying structure of the model ('vertically' partitioned). They also consider soft-partitioning approaches to overcome the limitations of disjoint partitioning. It is interesting future work to adapt such ideas for our problem, and compare with Staccato's approach.

Probabilistic databases have been studied in several recent projects (e.g., ORION [20],Trio [47], MystiQ [24], Sprout [43], and MayBMS [13]). Our work is complementary to these efforts: the queries we consider can produce probabilistic data that can be ingested by many of the above systems, while the above systems focus on querying restricted models (e.g., U-Relations or BIDs). We also use model-based views [25] to expose the results of query-time inference over the OCR transducers to applications.

The OCR, speech and IR communities have explored error correction techniques as well as approximate retrieval schemes [18, 28, 42]. However, prior work primarily focus on keyword search over plain-text transcriptions. Staccato can benefit from these approaches and is orthogonal to our goal of integrating OCR data into an RDBMS. In contrast, we advocate retaining the uncertainty in the transcription.

Many authors have explored indexing techniques for probabilistic data [33, 35, 38, 49]. Letchner et al. [38] design new indexes for RFID data stored in an RDBMS as Markovian streams. Kanagal et al. [33] consider indexing correlated probabilistic streams using tree partitioning algorithms and describe a new technique called *shortcut potentials* to speedup query answering. Kimura et al. [35] propose a new *uncertain primary index* that clusters heap files according to uncertain attributes. Singh et al. [49] consider indexing categorical data and propose an R-tree based index as well as a probabilistic inverted index. Our work focuses on the challenges that content models like OCR raise for integrating indexing with an RDBMS.

7 Conclusion and Future Work

We present our prototype system, Staccato, that integrates a probabilistic model for OCR into an RDBMS. We demonstrated that it is possible to devise an approximation scheme that trades query runtime performance for result quality (in particular, increased recall). The technical contributions are a novel approximation scheme and a formal analysis of this scheme. Additionally, we showed how to adapt standard text-indexing schemes to OCR data, while retaining more answers.

Our future work is in two main directions. Firstly, we aim to extend Staccato to handle larger data sets and more sophisticated querying (e.g., using aggregation with a probabilistic RDBMS, sophisticated indexing, parallel processing etc.). Secondly, we aim to extend our techniques to more types of content-management data such as speech transcription data. Interestingly, transducers provide a unifying formal framework for both transcription processes. Our initial experiments with speech data suggest that similar approximations techniques may be useful. This direction is particularly exciting to us: it is a first step towards unifying RDBMS and content-management systems, two multibillion dollar industries.

Acknowledgments

This work is supported by the National Science Foundation under IIS-1054009, the Office of Naval Research under N000141210041 and the Microsoft Jim Gray Systems Lab. Any opinions, findings, conclusions or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the US government or Microsoft. The authors also thank the anonymous

PVLDB reviewers as well as Jignesh Patel and Benny Kimelfeld for their valuable feedback on an earlier version of this paper.

References

[1] A Brief Introduction to Graphical Models and Bayesian Networks. http://www.cs.ubc.ca/ murphyk/Bayes/bayes.html.

[2] Condor high-throughput computing system. http://www.cs.wisc.edu/condor/.

[3] Content Management Systems. http://www.cmswire.com/.

[4] Corncob List. http://www.mieliestronk.com/wordlist.html.

[5] Digital humanities by UW's Prof. Witmore. http://winedarksea.org.

[6] ExperVision Inc. http://www.expervision.com/.

[7] Google Books. http://books.google.com/.

[8] OCRopus open source OCR system. http://code.google.com/p/ocropus.

[9] The Hathi Trust. http://www.hathitrust.org/.

[10] The JSTOR Archive. http://www.jstor.org/.

[11] Cyril Allauzen, Mehryar Mohri, and Murat Saraclar. General indexation of weighted automata - application to spoken utterance retrieval. In *Workshop on Interdisciplinary Approaches to Speech Indexing and Retrieval (HLT/NAACL)*, pages 33–40, 2004.

[12] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. Openfst: A general and efficient weighted finite-state transducer library. In *CIAA*, pages 11–23, 2007.

[13] Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.

[14] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[15] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, Inc., 2006.

[16] Vincent D. Blondel and John N. Tsitsiklis. When is a pair of matrices mortal? *Inf. Process. Lett.*, 63(5), 1997.

[17] Olivier Bournez and Michael S. Branicky. The mortality problem for matrices of low dimensions. *Theory of Computing Systems*, 2002.

[18] James Callan, W. Bruce Croft, and Stephen M. Harding. The inquery retrieval system. In *DEXA*, pages 78–83, 1992.

[19] M. Y. Chen, A. Kundu, and J. Zhou. Off-line handwritten word recognition using a hidden markov model type stochastic network. *Pattern Anal. Mach. Intell.*, 16:481–496, May 1994.

[20] Reynold Cheng, Dmitri V. Kalashnikov, and Sunil Prabhakar. Evaluating probabilistic queries over imprecise data. In *SIGMOD*, pages 551–562, 2003.

[21] Junghoo Cho and Sridhar Rajagopalan. A fast regular expression indexing engine. In *ICDE*, pages 419–430, 2001.

[22] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *STOC*, pages 91–100, 2004.

[23] Robert G. Cowell, A. Philip Dawid, Steffen L. Lauritzen, and David J. Spiegelhalter. *Probabilistic Networks and Expert Systems: Exact Computational Methods for Bayesian Networks.* Springer, 2007.

[24] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

[25] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.

[26] Jr. Forney, G.D. The viterbi algorithm. *Proc. IEEE*, 61:268–278, 1973.

[27] Rahul Gupta and Sunita Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.

[28] S.M. Harding, W. B. Croft, and C. Weir. Probabilistic retrieval of ocr degraded text using n-grams. In *ECDL*, pages 345–359, 1997.

[29] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., 2006.

[30] F.V. Jensen and S.K. Andersen. Approx. in bayesian belief universes for knowledge-based systems. In *UAI*, pages 162–169, 1990.

[31] M. I. Jordan. *Learning in graphical models.* MIT Press, 1999.

[32] Bhargav Kanagal and Amol Deshpande. Efficient query evaluation over temporally correlated probabilistic streams. In *ICDE*, pages 1315–1318, 2009.

[33] Bhargav Kanagal and Amol Deshpande. Indexing correlated probabilistic databases. In *SIGMOD*, pages 455–468, 2009.

[34] Benny Kimelfeld and Christopher Ré. Transducing markov sequences. In *PODS*, pages 15–26, 2010.

[35] Hideaki Kimura, Samuel Madden, and Stanley B. Zdonik. Upi: A primary index for uncertain databases. *PVLDB*, 3(1):630–637, 2010.

[36] Arun Kumar and Christopher Ré. Probabilistic management of ocr data using an rdbms. *UW-CS-Technical Report*, 2011. available from `http://www.cs.wisc.edu/hazy/staccato/papers/HazyOCR_TR.pdf`.

[37] John Lafferty. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. Morgan Kaufmann, 2001.

[38] Julie Letchner, Christopher Ré, Magdalena Balazinska, and Matthai Philipose. Access methods for markovian streams. In *ICDE*, pages 246–257, 2009.

[39] Julie Letchner, Christopher Ré, Magdalena Balazinska, and Matthai Philipose. Approximation trade-offs in markovian stream processing: An empirical study. In *ICDE*, pages 936–939, 2010.

[40] S. E. Levinson, L. R. Rabiner, and M. M. Sondhi. An introduction to the application of the theory of probabilistic functions of a markov process to automatic speech recognition. *Bell Systems Technical Journal*, 62:1035–1074, 1983.

[41] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[42] Shunji Mori, Hirobumi Nishida, and Hiromitsu Yamada. *Optical character recognition.* John Wiley & Sons, Inc., 1999.

[43] Dan Olteanu, Jiewen Huang, and Christoph Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, pages 640–651, 2009.

[44] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proc. of IEEE*, pages 257–286, 1989.

[45] Christopher Ré, Julie Letchner, Magdalena Balazinska, and Dan Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, pages 715–728, 2008.

[46] Christopher Ré and Dan Suciu. Approximate lineage for probabilistic databases. *PVLDB*, 1(1):797–808, 2008.

[47] Anish Das Sarma, Omar Benjelloun, Alon Halevy, and Jennifer Widom. Working models for uncertain data. ICDE, pages 7–18, 2006.

[48] Anish Das Sarma, Martin Theobald, and Jennifer Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, pages 1023–1032, 2008.

[49] Sarvjeet Singh, Chris Mayfield, Sunil Prabhakar, Rahul Shah, and Susanne Hambrusch. Indexing uncertain categorical data. In *ICDE*, pages 616–625, 2007.

[50] Paavo Turakainen. Generalized automata and stochastic languages. *Proc. of American Mathematical Society*, 21(2), 1969.

[51] Martin J. Wainwright and Michael I. Jordan. Graphical models, exponential families, and variational inference. *Foundations and Trends of Machine Learning*, 1, 2008.

[52] Daisy Zhe Wang, Eirinaios Michelakis, Minos N. Garofalakis, and Joseph M. Hellerstein. Bayesstore: managing large, uncertain data repositories with probabilistic graphical models. *PVLDB*, 1(1):340–351, 2008.

[53] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[54] Jin Y. Yen. Finding the k shortest loopless paths in a network. In *Management Science*, 1971.

[55] Justin Zobel, Alistair Moffat, and Ron Sacks-davis. An efficient indexing technique for full-text database systems. In *VLDB*, 1992.

[56] Argyrios Zymnis, Stephen Boyd, and Dimitry Gorinevsky. Relaxed maximum a posteriori fault identification. *Signal Process.*, 89, June 2009.

A Finite State Transducers

As mentioned in Section 2, the formal model used by STACCATO to encode the uncertainty information in OCR data is the Finite State Transducer (FST). A transducer is an automaton that converts (*tranduces*) strings from an input alphabet to an output alphabet. We can view a transducer as an SFA that both reads and emits characters on its transitions. Formally, we fix an input alphabet Γ and an output alphabet Σ. An FST S over Γ and Σ is a tuple $S = (V, E, s, f, \delta)$ where V is a set of nodes, $E \subseteq V \times V$ is a set of edges such that (V, E) is a directed acyclic graph, and s (resp. f) is a distinguished start (resp. final) node (state). Each edge has finitely many arcs. The function δ is a stochastic transition function, i.e.,

$$\delta : E \times \Gamma \times \Sigma \to [0,1] \text{ s.t. } \sum_{\substack{y:(x,y)\in E \\ \gamma\in\Gamma,\sigma\in\Sigma}} \delta((x,y),\gamma,\sigma) = 1 \quad \forall x \in V$$

In essence, $\delta(e, \gamma, \sigma)$, where $e = (x, y)$, is the conditional probability of transitioning from $x \to y$, reading γ and emitting σ. In OCR, the input alphabet is an encoding of the location of the character glyphs in the image, while the output alphabet is the set of ASCII characters. An FST also defines a discrete probability distribution over strings through its outputs.

B Illustrations for FindMinSFA

We now present more illustrations for the FindMinSFA operation (Section 3.1) in Figure 12. As shown in Algorithm 1, three cases arise when the given subset of nodes of the SFA S do not form an SFA by themselves. Firstly, they might not have a unique start node, in which case their least common ancestor has to be computed (Figure 12 (A)). Secondly, they might not have a unique end node, in which case their greatest common descendant has to be computed (Figure 12 (B). Finally, there could be an external edge incident upon an internal node of the subset (Figure 12 (C)). In all cases, FindMinSFA outputs a subset of nodes that form a valid SFA, which is then collapsed and replaced with a single edge in S.

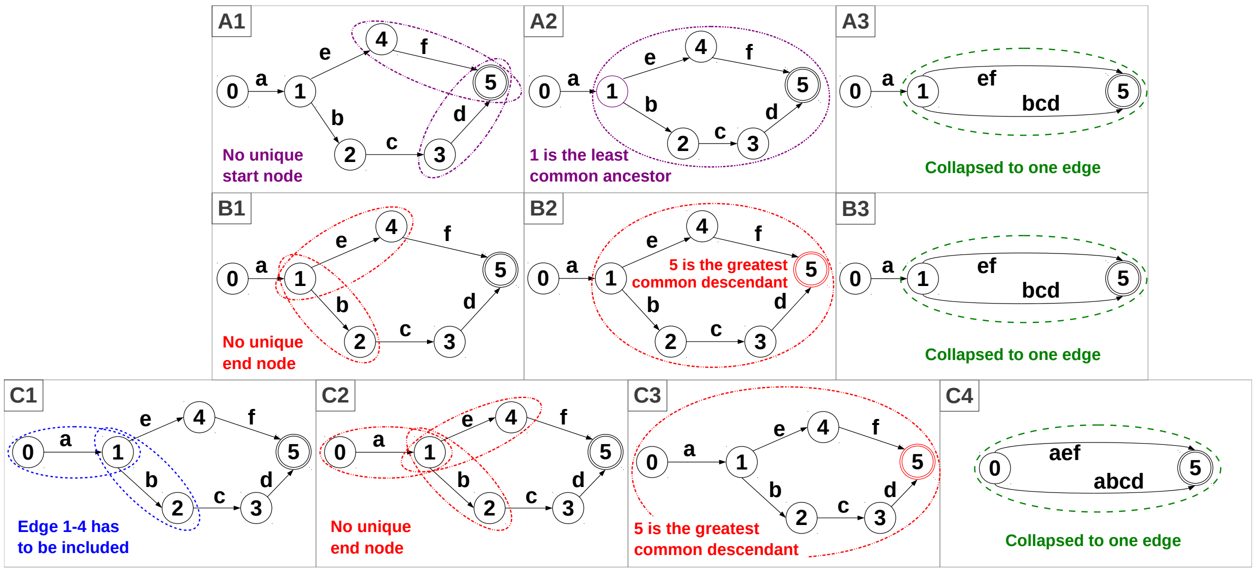


Figure 12: Illustrating FindMinSFA: (A) No unique start node for the set $X = \{3, 4, 5\}$, (B) No unique end node for the set $X = \{1, 2, 4\}$, and (C) Set $X = \{0, 1, 2\}$ has external edge $1-4$ incident on internal node 1, and has to be included.

C Conditional is a KL Minimizer

KL divergence is similar to a distance metric in that it allows us to say whether or not two probability distributions are close. Given two probability distribution $\mu, \nu : \Sigma^* \to [0, 1]$ the KL-divergence is denoted $\mathsf{KL}(\mu||\nu)$ and is defined as:

$$\mathsf{KL}(\mu||\nu) = \sum_{\sigma\in\Sigma^*} \mu(x) \log \frac{\mu(x)}{\nu(x)}$$

The above quantity is only defined for μ, ν such that $\mu(x) > 0$ implies that $\nu(x) > 0$. If $\mu = \nu$ then $\mathsf{KL}(\mu||\nu) = 0$.

We justify our choice to retain the probability of each string we select by showing that it is in fact a minimizer for a common information theoretic measure, KL-divergence. Given a probability distribution μ on Σ^* and a set $X \subseteq \Sigma^*$, let $\mu_{|X}$ denote the result of conditioning μ on X. Let A be the set of all distributions on X. Then,

$$\mathsf{KL}(\mu_{|X}||\mu) \leq \min_{\alpha\in A} \mathsf{KL}(\alpha||\mu) \tag{1}$$

That is, selecting the probabilities according to the conditional probability distribution is optimal with respect to KL divergence. Eq. 1 follows from the observation that $\mathsf{KL}(\mu_{|X}||\mu) = -\log Z$ where $Z = \sum_{x\in X} \mu(x)$. Using the log-sum inequality one has

$$\sum_{x\in X} \alpha(x) \log \frac{\alpha(x)}{\mu(x)} \geq \left(\sum_{x\in X} \alpha(x)\right) \log \frac{(\sum_{x\in X} \alpha(x))}{(\sum_{x\in X} \mu(x))} = -\log Z$$

D δ_k is a minimizer (Proposition 3.1)

There are two observations. The first is that by normalization, since the probability of every string is simply proportional to its probability in $\Pr_{[\delta]}$ then the KL divergence is inversely proportional to the probability mass retained. Thus, the minimizer must retain as much probability mass as possible. The second observation is the following: consider any chunk (S_i, s, f) where s is the single start state and f is the final state. By construction, every path that uses a character from S_i must enter through s and leave through f. And the higher probability that we place in that state, the higher the retained mass. Since δ_k retains the highest probability in each segment, it is indeed the minimizer.

E Proof of Theorem 3.1

The starting point is that the following problem is NP-hard: Given vectors $x, y \in \mathbb{Q}^l$ and a fixed constant $\lambda \geq 0$ for $l = 4$ and sets of stochastic matrices $\mathcal{S}_1, \ldots, \mathcal{S}_N$ where each $\mathcal{S}_i$ is a set of 2 $l \times l$ matrices, determine if there is a sequence $\bar{i} \in \{1, 2\}^N$ such that $M_{i_j} \in \mathcal{S}_j$ and:

$$x^T M_{i_N} \cdots M_{i_1} y \geq \lambda$$

We find a small l such that the claim holds. For this, we start with the results of Bournez and Branicky who show that a related problem called the *Matrix Mortality problem* is NP-hard for matrices of size 2×2 [17], where we ask for to find a selection as above where $x^T M_{i_N} \cdots M_{i_1} y = 0$. Unfortunately, the matrices (M_{ij}) are not stochastic (not even positive). However, using the techniques of Turakainen [50] (and Blondel [16]), we can transform the matrices into slightly larger, but still constant dimensions, stochastic matrices ($l = 4$).

Now, we construct a transducer and chunk structure Φ such that if it is possible to choose at most $k = 2$ in each chunk with the total probability mass being greater than $\lambda 2^{-N}$, then we can get a choice for $\bar{i}$. Equally, if there exists such a choice for $\bar{i}$, then we can find such a transducer representation. So the problem of finding the highest mass representation is NP-hard as well.

Throughout this reduction, we assume that every edge is assigned a unique character to ensure the unique path property. It is straightforward to optimize for a binary alphabet: simply add replace each character a sequence of edges with a binary encoding (then make this one chunk). So, we omit the emitted string in the transition function.

Let $P(x)$ denote the probability mass that a string is emitted that passes through the node x. We will group nodes together as components of a vector. The start node s has $P(s) = 1$. Then, we construct the vector y by creating nodes $v_1, \ldots, v_l$ with a transition $\delta((s, v_i), 0) = y_i$. Thus, $P(v_i) = y_i$. We need two main gadgets: (1) A gadget to encode matrix multiplication and (2) a gadget that intuitively encodes that given two inputs, we can select one or the other, but not both. We provide a slightly weaker property: For a fixed parameter $\alpha \geq 0$ (we pick α below). We construct a gadget that takes as input two nodes x, x' and has output two nodes u, u' such that the probability at u ($P(u)$) and at u' ($P(u')$) satisfies the following weak-exclusivity:

$$(P(u), P(u')) = \big\{(P(x), 0), (0, P(x')), (v, v')\big\} \tag{2}$$

where $v \leq \alpha P(x)$ and $v' \leq \alpha P(x')$. Intuitively, this gadget forces us to choose x or x' or not both. Notice that if we select both, then for sure the output of each component is smaller than α.

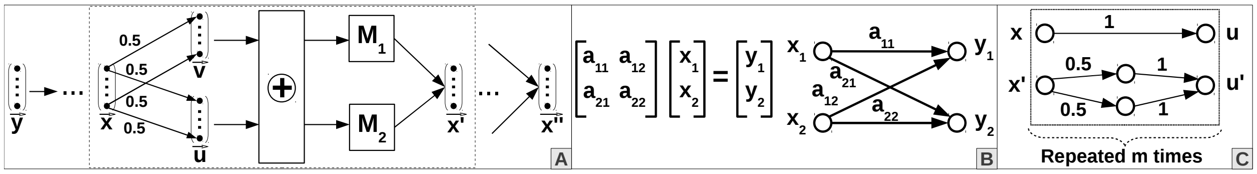


Figure 13: Gadgets used in the proof of Theorem 3.1: (A) Overall Reduction. We create one block for each of the N sets $\mathcal{S}_i$ (B) Multiply Gadget (C) Binary Exclusive Gadget

Assuming these gadgets, the overall construction is shown in Figure 13(A) that illustrates a chunk for a single set $\mathcal{S}_i$. Each chunk contains two matrix multiply gadgets (representing the two matrices in $\mathcal{S}_i$) and a large gadget that ensures we either choose one matrix or the other – but not elements of both. The input to the chunk is a vector x: in the first chunk, $x = y$ above. In chunk j, x will represent the result of some of choice $M_{i_{j-1}} \cdots M_{i_1} y$. As shown, for each $i = 1, \ldots, l$, we send x_i to v_i with probability 0.5 and x_i to u_i with probability 0.5. In turn, u is fed to the multiply gadget for M_{1j} and v is fed to the multiply gadget for M_{2j} (with $\delta = 1$). We ensure that we cannot select both v_i and u_j for any i, j using the exclusive gadget described below. The output of this chunk is either $0.5 M_{i1} x$ or $0.5 M_{i2} x$ or a vector with ℓ_1 norm smaller than α. We set $\alpha < 2^{-N}\lambda$. Given this, property is clear that given any solution to the original problem, we can create a solution to this problem. On the other hand, if the solution with highest probability mass has mass greater than $2^{-N}\lambda$ then it must be a valid solution (since we set $\alpha < 2^{-N}\lambda$). Now the gadgets:

The Multiply Gadget Matrix multiplication can be encoded via a transducer (see Fig. 13(B)). Notice that the "outputs" in the above gadget have the probability of the matrix multiply for 2×2 matrices. That is, given a matrix A and input nodes $x_1, \ldots, x_m$, the output nodes y_i above are such that $P(y_i) = \sum_{j=0}^{m} A_{ij} P(y_j)$. Each edge is a single chunk.

The Exclusive Gadget We illustrate the gadget for $k = 2$. We have two inputs x, x' and two outputs u, u'. Our goal is to ensure the property described by Eq. 2. The gadget is shown in Figure 13(C). The chunk here contains the entire gadget, since we can only select k paths, it is clear that each iteration of the gadget we get the property that:

$$(P(u), P(u')) = \{(P(x), 0), (0, P(x')), (v, v')\}$$

where $v \leq 0.5P(x)$ and $v' \leq 0.5P(x')$. Repeating the gadget m times (taking m s.t. $2^{-m} \leq 2^{-N}\lambda$ suffices). The property we need is that we only select one vector or the other – to ensure this we simply place $\binom{l}{2}$ gadgets: each one says that if $u_i > 0 \implies v_j = 0$ (and vice versa). Observe that the resulting gadget is polynomial sized. After concatenating the gadgets together, the end result is either a $2^{-N} M_{i_N} \cdots M_{i_i}$ (a valid result) or its ℓ_1 norm is smaller than $\lambda 2^{-N}$ (since it messes up on at least one of the gadgets).

Proof of Hardness We now complete the proof by showing that a problem, called StocAut is NP-hard – for a fixed size alphabet. Then, we show how to encode this in the Layout problem, thereby proving that Layout is NP-hard for a fixed size Σ.

The GenAut Generic automaton problem is the following: Given vectors $x, y \in \mathbb{Q}^k$ for some fixed k and sets of matrices $\mathcal{G}_1, \ldots, \mathcal{G}_N$ where each $\mathcal{G}_i$ for $i = 1, \ldots, N$ is a set of $k \times k$ matrices, the goal is to determine if there is a tuple of natural numbers $\bar{i}$ such that:

$$x^T M_{i_N} \cdots M_{i_1} y \geq 0 \text{ where } M_{i_j} \in \mathcal{G}_j$$

We are concerned with the related problem StocAut where all matrices and vectors in the problem are stochastic. A matrix is stochastic if all entries are positive, and its row sums and column sums are 1. A vector is stochastic if each entry is positive and its 1-norm (sum of entries) is 1. In StocAut the condition we want to check is slightly generalized:

$$x^T M_{i_N} \cdots M_{i_1} y \geq k^{-1} \text{ where } M_{i_j} \in \mathcal{G}_j$$

and k is the dimension of the problem.

Lemma E.1. *For fixed dimension, $k = 2$, the* GenAut *problem is NP-Complete in N even if $|\mathcal{G}_i| \leq 2$ for $i = 1, \ldots, N$.*

Proof. This is shown by using the observation that $x^T \bar{A} y = 0$ is NP-hard (exact sequence mortality problem [17]) and that $x^T \bar{A} y = 0$ if and only if $-x^T \bar{A} y x^T \bar{A} y \geq 0$. We then observe that yx^T is such a matrix. More precisely, the following problem is NP-hard: Given a set of values $s_1, \ldots, s_N$ is there a set $S \subseteq [N]$ such that $\prod_{i=1,\ldots,N} s_i = b$ for some fixed b.

$$H = \begin{pmatrix} 0 & 1 \\ 0 & -1 \end{pmatrix}$$

It is not hard to check that for any 2×2 A we have:

$$HAH = \begin{pmatrix} 0 & (a_{21} - a_{22}) \\ 0 & -(a_{21} - a_{22}) \end{pmatrix}$$

so that $HAH = 0$ if and only if $a_{21} = a_{22}$. Then, we create the following matrices:

$$S_i = \begin{pmatrix} 1 & 0 \\ 0 & s(A) \end{pmatrix} \text{ and } B = \begin{pmatrix} 1 & 0 \\ b & 1 \end{pmatrix}$$

Then, denote by I_2 the identity matrix. Then, we set: $\mathcal{G}_0 = \{HB\}$ and $\mathcal{G}_i = \{S_i, I_2\}$ for $i = 1, \ldots, N$. The vectors are $x = (1, 1)$ and $y = x^T$. Then, applying the construction above proves the claim. □

We now apply Turakainen's technique [50] (we learned of the technique from Blondel [16]) to transform the above matrices into slightly larger, but still constant dimensions, that are positive and then finally stochastic. First, we define a further restriction ZeroAut which requires that each matrix row/column sum is zero. We prove that this is still NP-complete over slightly larger matrices.

Lemma E.2. *For dimension $k = 4$, the* ZeroAut *problem is NP-complete.*

Proof. For each matrix M above we create a new 4×4 matrix N as follows:

$$N = \begin{pmatrix} 0 & \bar{0} & 0 \\ \alpha(x) & M & 0 \\ \beta_0(x) & \beta(x) & 0 \end{pmatrix}$$

we choose α, β to be vectors such that the row,column sums are zero. Then, we take the original x and create $x_1 = (0, x, 0)$ and $y_1 = (0, y^T, 0)^T$. It follows then that for any set of matrices:

$$x_1 \bar{N} y_1^T = xMy^T$$

Thus, we have shown the stronger claim that all products are equal and so it directly follows that the corresponding decision problem is NP-complete. $\square$

We define StocAut to be the restriction that all matrices are stochastic (as above) and we check a slightly generalized condition:

$$x^T M_{i_N} \cdots M_{i_1} y \geq k^{-1}$$

where $M_{i_j} \in \mathbb{R}^k$.[7]

Lemma E.3. *The problem* StocAut *is NP-complete for matrices of* 4×4.

Proof. We first show that we may assume that the vectors from the ZeroAut problem are stochastic. Let $\mathbf{1}$ be the all ones vector in $\mathbb{R}^k$. Observe that for any Z such that the row and column sums are 0 (as is each matrix in the input of ZeroAut) then:

$$(x + \alpha\mathbf{1})^T Z (y + \alpha\mathbf{1}) = x^T Z y$$

for any value of α since $\mathbf{1}^T Z = Z\mathbf{1} = \mathbf{0}$ the zero vector. Take $\alpha = |\min_i \min_{z=x,y} z_i|$. Thus, we can take $x + \alpha\mathbf{1}$ and $y + \alpha\mathbf{1}$ and preserve the product (which are both entry-wise positive). Then, we can scale both $x + \alpha\mathbf{1}$ and $y + \mathbf{1}$ by any positive constant:

$$(\alpha x)^T Q (\beta y) = \alpha\beta x^T Q y$$

So the sign is preserved if $\alpha, \beta > 0$. And we can assume without loss of generality that both x, y are stochastic by scaling by their respective ℓ_1 norms. (If either $\|x\|_1 = 0$ or $\|y\|_1 = 0$ the problem is trivially satisfied: the product is 0 and since they must be the zero vector.)

We now show how to achieve the condition of stochastic matrices. First, we show how to achieve positive matrices. To do this, let Q be the all ones matrix of 4×4 ($Q_{ij} = 1$). Then let $\lambda \geq 0$ be such that for all matrices $M + \lambda Q \geq 0$ entrywise. Then, for each matrix M replace it with a new matrix N defined as:

$$N = (\lambda k)^{-1} (M + \lambda Q)$$

Since $MQ = 0$, the following holds:

$$x^T \prod_{i=1}^{N} (M_i + \lambda Q) y = (\lambda k)^{-m} x^T \prod_{i=1}^{N} M_i y + x^t Q y$$

Thus, the original product is positive if and only if the modified product is bigger than k^{-1} proving the claim. $\square$

[7]Although unnecessary for our purpose we observe that with r repetitions of the problem, one can drive the constant down to k^{-r}. Thus, the constant here is arbitrary.

F Algorithms

Notations:
$G = (V, E, s, f, \delta)$, the Staccato data SFA
$Q = (V_Q, E_Q, s_Q, F_Q, \delta_Q)$, the dictionary DFA
$D = \{(f_Q \in F_Q, term)\}$, the dictionary terms (hash table)
$AugSts = \{(v_Q \in V_Q, PostingSet)\}$, augmented states (hash table)
$I = \{(term, PostingSet)\}$, the index (hash table)

Figure 14: Notations for Algorithms 3 and 4

Here we present the algorithm for constructing the inverted index for a given SFA, as referred to in Section 4. The notations used are listed in Figure 14.

Algorithm 3: The dynamic program for Staccato index construction

```
$\forall e \in E$ with parent edges $e' \in E$
    $\forall f_Q \in F_Q,\ AugSts_{par}(f) = \cup_{e'} AugSts_{e'}(f_Q)$
    $AugSts_e = RunDFA(AugSts_{par}, e)$
otherwise,
    $\forall f_Q \in F_Q,\ AugSts_e(f_Q) = \phi$
```

The construction, presented in Algorithms 3 and 4, is similar to automata composition. The dictionary of terms is first compressed into a trie-automaton [29] with multiple final states, each corresponding to a term. Then, we walk through the data SFA (using a dynamic program on the SFA's graph) and obtain the locations (postings) where any dictionary term starts. A key thing to note here is that terms can straddle mutiple SFA edges, which needs to be tracked. We pass information about such multi-edge terms through sets of 'augmented states', which store pairs of the query DFA's state and possible postings. When the DFA reaches a final state (i.e., a term has been seen), the corresponding postings are added to the index.

G Implementation Details

Each line of a document corresponds to one transducer, which is stored as such in the FullSFA approach. k-MAP stores a ranked list of strings for each line after inference on the transducer. In Staccato, each line corresponds to a graph of chunks, where each chunk is a ranked list of strings. These data are stored inside the RDBMS with a relational schema, shown in Table 5. There is one master table per dataset, which contains the auxiliary information like document name, line number, etc., and there are separate data tables for each approach.

Algorithm 4: *RunDFA(AugSts,e)*

```
for each string $p_i$ on $e$, $i = 0$ to $k-1$ do
    $SO \leftarrow \{(0,0)\}$ //{(State, Offset)}
    for each character $c_j$ in $p$, $j = 0$ to $|p|-1$ do
        $NSO \leftarrow \phi$
        for each $t \in SO$ do
            $Nxt \leftarrow \delta_Q(t.State, c_j)$
            if $Nxt \neq 0$ then
                $NSO \leftarrow NSO \cup (Nxt, t.Offset)$
                if $Nxt \in F_Q$ then
                    $I(D(Nxt)) \leftarrow I(D(Nxt)) \cup (e, i, t.Offset)$
        $SO \leftarrow NSO \cup \{(0, j+1)\}$
    for each $r \in SO$ do
        if $r.State \neq 0$ then
            $NAugSts(r.State) \leftarrow NAugSts(r.State) \cup \{(e, i, r.Offset)\}$
    for each $d \in AugSts$ do
        $Cur \leftarrow d.State$
        for each character $c_j$ in $p$, $j = 0$ to $|p|-1$ do
            $Nxt \leftarrow \delta(t.State, c_j)$
            if $Nxt \neq 0$ then
                $Cur = Nxt$
                if $Nxt \in F_Q$ then
                    for each $l \in d.PostingSet$ do
                        $I(D(Nxt)) \leftarrow I(D(Nxt)) \cup \{l\}$
            else
                break //DFA 'dies' reading string
            if $j = |p|-1$ then
                for each $l \in d.PostingSet$ do
                    $NAugSts(Cur) \leftarrow NAugSts(Cur) \cup \{l\}$
```

Approach	Table Name	Attributes		Primary Key
		Name	Type	
-	MasterData	DataKey	INTEGER	DataKey
		DocName	VARCHAR(50)	
		SFANum	INTEGER	
k-MAP	kMAPData	DataKey	INTEGER	DataKey, LineNum
		LineNum	INTEGER	
		Data	TEXT	
		LogProb	FLOAT8	
FullSFA	FullSFAData	DataKey	INTEGER	DataKey
		SFABlob	OID	
Staccato	StaccatoData	DataKey	INTEGER	DataKey, ChunkNum, LineNum
		ChunkNum	INTEGER	
		LineNum	INTEGER	
		Data	TEXT	
		LogProb	FLOAT8	
	StaccatoGraph	DataKey	INTEGER	DataKey
		GraphBlob	OID	
-	GroundTruth	DataKey	INTEGER	DataKey
		Data	TEXT	

Table 5: Relational schema for storing SFA data

H Extended Experiments

We now present more experimental results relating to runtimes and answer quality for the filescans, as well as some aspects of the inverted indexing. The queries we use are listed in Table 6, along with the number of ground truth answers for each on their respective datasets.

H.1 Recall and Runtime

Table 7 presents the precision and recall results of the queries, while Table 8 presents the respective runtime results. The values of the parameters are $m = 40$, $k = 50$ *and* $NumAns = 100$. As in Section 5, here too we see that Staccato lies between k-MAP and FullSFA on both recall and runtime. The precision too exhibits a similar trend. Again, the FullSFA approach is upto three orders of magnitude slower than MAP but achieves perfect recall on most queries, though precision is lower. Interestingly, on some queries in the DB dataset (e.g., DB3 and DB6), Staccato recall is close to 1.0 while the runtime is about two orders of magnitude lower than FullSFA. Another thing to note is that the recall increase for k-MAP and Staccato (over MAP) is more pronounced in DB and LT than CA. We can also see that keyword queries can have lower recall than some regex queries (e.g., LT3 and DB2).

Dataset	S.No.	Query	# in Truth
CA	1	Attorney	28
	2	Commission	128
	3	employment	73
	4	President	14
	5	United States	52
	6	Public Law (8|9)$\backslash d$	55
	7	U.S.C. 2$\backslash d\backslash d\backslash d$	25
DB	1	accuracy	65
	2	confidence	36
	3	database	43
	4	lineage	83
	5	Trio	68
	6	Sec$(\backslash x) * d$	33
	7	$\backslash x\backslash x\backslash x\backslash d\backslash d$	47
LT	1	Brinkmann	92
	2	Hitler	12
	3	Jonathan	18
	4	Kerouac	21
	5	Third Reich	7
	6	19$\backslash d\backslash d,\ \backslash d\backslash d$	32
	7	$spontan(\backslash x)*$	99

Table 6: Queries and ground truth numbers

Query	Approach			
	MAP	k-MAP	FullSFA	STACCATO
CA1	1.00/0.93	1.00/0.93	0.28/1.00	0.87/0.96
CA2	1.00/0.78	1.00/0.78	1.00/0.78	1.00/0.78
CA3	1.00/0.90	1.00/0.90	0.73/1.00	0.97/0.93
CA4	1.00/0.79	1.00/0.79	0.14/1.00	0.85/0.79
CA5	1.00/0.77	1.00/0.79	0.52/1.00	1.00/0.88
CA6	1.00/0.87	1.00/0.96	0.55/1.00	1.00/0.98
CA7	1.00/0.28	1.00/0.52	0.25/1.00	0.50/0.80
DB1	1.00/0.58	0.98/0.93	0.65/1.00	0.95/0.97
DB2	0.00/0.00	0.87/0.19	0.36/1.00	0.90/0.53
DB3	0.85/0.67	0.87/0.79	0.43/1.00	0.90/1.00
DB4	0.97/0.91	0.96/0.93	0.82/0.99	0.85/0.95
DB5	0.93/0.75	0.90/0.95	0.67/0.99	0.79/0.96
DB6	0.96/0.76	0.96/0.81	0.33/1.00	0.40/0.96
DB7	0.91/0.85	0.73/0.89	0.44/0.94	0.42/0.89
LT1	0.96/0.87	0.96/0.90	0.92/1.00	0.94/0.91
LT2	1.00/0.92	1.00/1.00	0.12/1.00	0.12/1.00
LT3	1.00/0.11	1.00/0.17	0.18/1.00	0.94/0.83
LT4	0.81/0.62	0.86/0.90	0.21/1.00	0.74/0.95
LT5	1.00/0.29	1.00/1.00	0.07/1.00	1.00/1.00
LT6	0.77/0.65	0.76/0.67	0.31/0.97	0.26/0.81
LT7	0.84/0.88	0.83/0.88	0.83/0.88	0.83/0.88

Table 7: Precision and recall results

Query	Approach			
	MAP	k-MAP	FullSFA	Staccato
CA1	0.17	0.82	81.54	4.38
CA2	0.17	0.96	91.84	4.93
CA3	0.17	0.96	91.85	4.94
CA4	0.17	0.89	86.72	4.63
CA5	0.18	1.16	106.17	5.97
CA6	0.18	1.17	125.63	5.98
CA7	0.18	1.05	150.35	5.40
DB1	0.07	0.44	56.42	1.61
DB2	0.07	0.51	62.89	1.81
DB3	0.07	0.43	54.92	1.59
DB4	0.07	0.40	51.45	1.48
DB5	0.07	0.42	40.72	1.21
DB6	0.07	0.35	619.31	1.39
DB7	0.07	0.31	1738.78	1.37
LT1	0.14	0.73	83.78	3.27
LT2	0.13	0.59	69.68	2.72
LT3	0.14	0.71	79.76	3.10
LT4	0.14	0.65	74.58	2.90
LT5	0.14	0.85	93.35	3.72
LT6	0.14	1.02	155.45	4.52
LT7	0.15	1.00	887.19	4.23

Table 8: Runtime results. Runtimes are in seconds.

H.2 Precision and F-1 Score

Though our focus is on recall-sensitive applications, we also study how the precision is affected when we vary m and k. For the same queries and parameter setting as in Figure 6, we plot the precision and F-1 score of the answers obtained. Figure 15 shows the results.

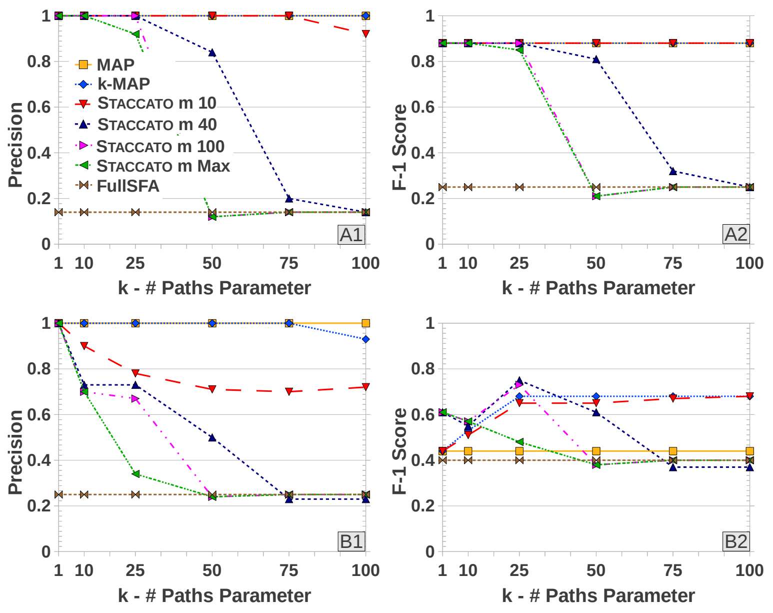


Figure 15: Precision and F-1 Score variations with k on two queries: (A) '*President*', and (B) '*U.S.C.* 2\d\d\d'. *NumAns* is set to 100.

As mentioned before, k-MAP precision is high, since it returns only a few answers, almost all of which are correct. On the other hand, FullSFA precision is lowest, since it returns many incorrect answers (along with most of the correct ones). Again, STACCATO falls in between, with the precision being high (close to k-MAP) for lower values of m and k, and gradually drops as we increase m and k. Also, the precision drops faster for higher values of m. It should be noted that the precision needn't drop monotonically, since additional correct answers might be obtained at higher m and k, boosting both the recall and precision. For completeness sake, the F-1 score variation is also presented in Figure 15. Interestingly, for the regex query, the F-1 score of both k-MAP and FullSFA are lower than that of STACCATO, the former due to its lower recall, and the latter due to its lower precision.

H.3 Sensitivity to NumAns

As we mentioned in Section 5, the quality of the answers obtained is sensitive to the *NumAns* parameter. If it is set too low (lower than the number of ground truth answers), the recall is likely to be low. On the other hand, if it is set too high, the recall will increase, but the precision might suffer. Thus, we perform a sensitivity analysis on *NumAns* for the recall, precision and F-1 score obtained. The same queries as in Figure 6 are used and the parameter setting is $m = 40,\ k = 75$. Figure 16 shows the results.

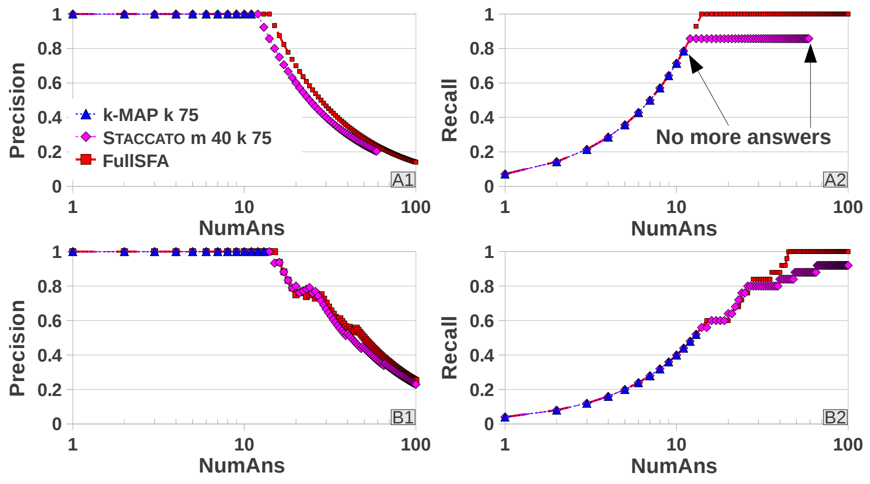


Figure 16: Sensitivity of Precision and Recall to *NumAns* on two queries: (A) *President*, and (B) *U.S.C.* $2\backslash d\backslash d\backslash d$. The x-axes are in logscale.

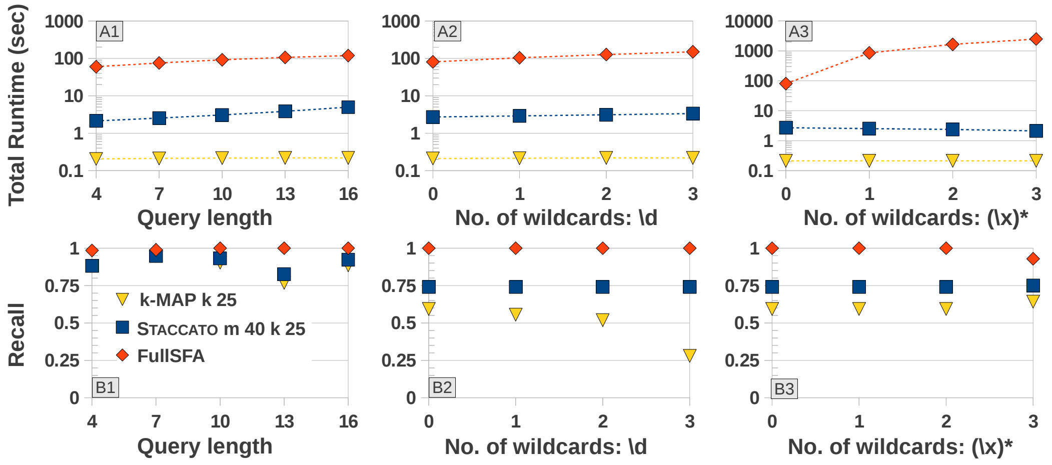


Figure 17: Impact of Query Length and Complexity: (A1, B1) correspond to the keyword queries, (A2, B2) correspond to the simple wildcard queries, while the last pair correspond to the complex wildcard queries. *NumAns* is set to 100. Runtimes are in logscale.

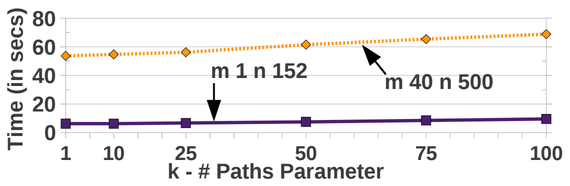


Figure 18: Sensitivity of STACCATO construction time to the parameter k

As Figures 16 (A1) and (A2) show, the precision initially remains high (here, at 1) when $NumAns$ is low. This is because the highest probability answers (that appear on top) are likely to be correct. As we increase $NumAns$, we get more correct answers and thus recall increases constantly. Once we near the number of ground truth answers, the recall starts to flatten, while the precision starts to drop. For k-MAP, beyond a value of $NumAns$, no more answers are returned since no matches exist. On the other hand, in FullSFA, almost all SFAs match almost all queries, and so we keep getting answers. Moreover, since FullSFA uses the full probabilities, the increase is relatively smooth. STACCATO for the given parameters gives more answers than k-MAP, and achieves higher recall but falls short of FullSFA. The recall for the regex query in STACCATO recall keeps increasing after being relatively flat many times. Thus, STACCATO can still achieve high recall, though at a lower precision than FullSFA.

H.4 Effect of Query Length and Complexity

We now study the impact of they query length and complexity on the runtime and recall obtained. For this study, we use three sets of queries. The first set consists of keyword queries of increasing length. The second set consists of regular expression queries with an increasing number of simple wildcards, e.g., '$U.S.C.\ 2\backslash d\backslash d$' (where $\backslash d$ is any digit). The third set too consists of regular expression queries, but with the more complex Kleene star as wildcards, e.g., '$U(\backslash x) * S(\backslash x) * C.\ 2$' (where $\backslash x$ is any character). Figure 17 shows the runtime and recall results for these queries.

As expected, the plots show that the runtime increases polynomially, but slowly with query length in all the approaches. However, the increase is more pronounced for FullSFA with complex wildcards (Figure 17:A3) since the composition based query processing produces much larger intermediate results. It can also be seen that there is no definite trend in the obtained recall for (Figure 17:A1), since a longer query can have better recall than a shorter one.

H.5 Staccato Construction

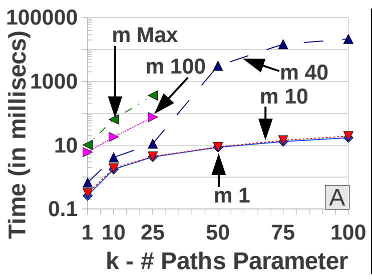
We now present the sensitivity of the STACCATO construction times to the parameter k. Figure 18 shows the results.

The plot shows that the runtimes increase linearly with k. However, as mentioned in Section 5, this linearity is not guaranteed since the chunk structure obtained may not be the same across different values of k, for a fixed SFA and m.

H.6 Index Construction Time

Here, we discuss the runtimes for the STACCATO index construction, which is a two-phase process. First, we obtain the postings independently for each SFA, and then unify all postings into the index. We pick a few SFAs and run the indexing in a controlled setting.

Figure 19 shows the sensitivity of the construction times for m and k for a single SFA, and also tabulates the bulk index load times for an entire dataset (LT). Firstly, we can see these runtimes are mostly practical. Also, we can see a linear trend in k, with a non-linearity showing up at $m = 40, k = 50$. we found that this was because the data in this parameter space had many single-character wide blocks, leading to the presence of more terms, and blowing up the size of the index. This causes two effects - the number of postings per SFA goes up by upto three orders



k \ m	1	10	40	100	Max
1	8	25	29	33	33
10	29	32	43	60	97
25	43	43	58	106	303
50	66	67	1,497	-	-
75	96	99	7,365	-	-
100	111	126	10,690	-	- B

Figure 19: (A) STACCATO index construction times. Note the logscale on the y-axis. (B) Bulk index load times (in seconds) for the index tables of the LT dataset.

of magnitude, and the selectivity of most terms across the dataset shoots up too, as was seen in Figure 9 (A).

Since running the indexing on all SFAs is also easily parallelizable, we again used Condor [2]. Overall, the index construction on all the data, for the above parameters took about 3 hours. After obtaining the postings lists for all SFAs, we loaded them all into the index table. These bulk load times are tabulated in Figure 19 (B). We can see that the load times are concomitant with the construction times due to the size of the index obtained.

H.7 Index Utility and Size

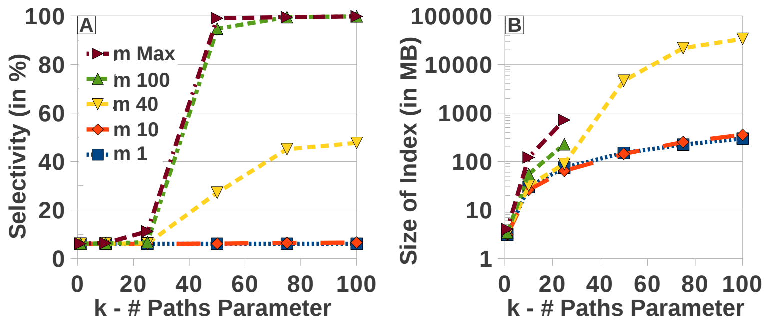


Figure 20: (A) Selectivity (%ge of SFAs) of the term '*public*' using the STACCATO index, for various values of m and k. (B) Size of the STACCATO index. Note the logscale on the y-axis.

It was discussed in Section 5 that the inverted index becomes less 'useful' as m and k become higher. To justify that, we study the selectivity of a query that uses the index. Here, we define selectivity as the percentage of the SFAs in the dataset that match the query when using the index. Figure 20 (A) shows the results for a query on the CA dataset. A complementary aspect of the utility of the index is its size. Figure 20 (B) shows the size of the index over the data in STACCATO.

Two interesting things can be seen from these plots. The query selectivity for lower values of m and k is relatively low, but for middle values of m ($m = 40$), the term starts to appear in many more SFAs as k increases. For higher values of m ($m = 100, Max$), as k increases, the the selectivity shoots up to nearly 100%, which means that almost all SFAs in the dataset contain the term. This implies that the index is no longer useful in the sense that almost the entire dataset is returned as answers. We observed the same behavior across all queries and datasets. The plot of the index sizes reflects this phenomenon. The size varies largely linearly as expected, but at $m = 40, k = 50$, it shoots up two orders of magnitude, similar to Figure 19. This size increase is largely because of the selectivity increase, i.e., many more entries appearing in the index. The index construction for $m = 100, Max$ for $k = 50$, and above was skipped since the index sizes exceeded the available disk space (over 200 GB). However, the selectivity can be easily computed after obtaining just the first posting, with no need to compute all the postings. The selectivity confirms that these parameter settings do not give useful indexes anyway.