A Taxonomy of Query Auto Completion Modes

Unni Krishnan
The University of Melbourne
Melbourne, Australia

Alistair Moffat
The University of Melbourne
Melbourne, Australia

Justin Zobel
The University of Melbourne
Melbourne, Australia

ABSTRACT

Query auto completion mechanisms assist users to formulate search requests by suggesting possible queries corresponding to incomplete text they have typed. Keystroke by keystroke, these mechanisms proceed by finding matching strings from resources such as logs that have captured the behavior of previous users; they might also be informed by key phrases extracted from indexed documents, including, for example, anchor-text strings. Here we explore the range of ways, or modes, in which strings might be thought of as "matching" a partially typed query, and hence develop a taxonomy of possible approaches, each requiring different implementation structures and algorithms. Past work in the field has typically focused on only one or another of the modes, creating a lack of clarity as to exactly what challenge is being addressed. We use our taxonomy to survey options for auto completion and provide preliminary measurements in regard to their computational cost, using a range of public implementations.

KEYWORDS

Query auto completion; query log; string search

1 INTRODUCTION

Query auto completion (QAC) is a feature found in many search services, including Google, Bing, Facebook, Twitter, and Wikipedia. Query completions assist the user to formulate their search request. They reduce the number of keystrokes required to enter the query that was intended, assist by spelling the remainder of a search term, and provide guidance as to what the query might become; this last is an informative function in its own right, as it may be that the wording of the query was the knowledge being sought. Generation of a list of suggested completions involves searching over a collection of strings (often, for example, the user's prior queries or the most popular queries that had been presented to the search service) to find those that match best against the current partial query string that has been entered.

Traditional search systems display summary information describing target documents – each of which is typically dozens, hundreds, or thousands of words long – in response to a *full* query string. In contrast, QAC systems work with a partial queries at lengths measured in handfuls of characters, and must respond with "answers" that are themselves usually only a few words long. As each character is typed, the list of answers is recalculated; the user then either selects one of the proffered suggestions or ignores them and continues to extend their query.

The usefulness of a QAC system depends on both the efficiency with which it computes the completions and the quality of the proposed suggestions. Users expect completions that are plausible expansions of the query they are intending to submit, and expect them to be generated and displayed without any noticeable delays. The tension between these two objectives makes the design of QAC systems inherently challenging, and a good design must acknowledge both of these requirements. Another important aspect is the order in which the completions are displayed to the user. Typically, the top-k highest-scoring completions based on a similarity function are displayed, in descending score order. The mechanics of calculation of this score is another aspect that governs the performance of QAC systems.

There are several ways in which user-entered partial queries might be interpreted. A common mechanism is to assume that the partial text is a prefix, and, perhaps, if there are spaces present then to assume that each partial (alphabetic) string is a prefix of a different term in the desired query; however, in the interests of having an inclusive taxonomy, we do not assume a particular matching mechanism. Instead, we describe the partial text as a pattern, and explore different ways in which the partial text and a complete query might be matched.

We use P to denote the current query pattern, and note that it is dynamic, with characters appended (and also sometimes removed) as the user types. The aim of a QAC system is to compute a ranked list of completions that extrapolates P in a useful manner. A common starting point is a log of previous completed queries and their frequencies, used as a reference as to what users are (most recently) searching for. In this context, the steps involved in a QAC system can be summarized as:

(1) Retrieve the set of candidates in the set of possible target strings that match the pattern P provided by the user;
(2) Rank those candidates by their frequency, either in conjunction with Step 1, or through the application of a separate ranking operation;
(3) Optionally re-rank an initial subset of the sorted candidate list, based on a second more complex ranking criterion, such as predicted popularity, search context, personalization concerns, and so on; and
(4) Present the top-k suggestions to the user in a manner that allows them to select one, or to continue typing, or to select one and then continue typing.

A range of methods can be used to sort the completions and determine a top-k, for steps (2) and (3). Here our focus is on the matching process itself, step (1).

ADCS 2017, December 7–8, 2017, Brisbane, QLD, Australia

ACM 978-1-4503-6391-4/17/12…$15.00
https://doi.org/10.1145/3166072.3166081

Symbol	Definition
Σ	alphabet for P and strings in T
P	current query pattern
$P^1, P^2, \ldots, P^m$	tokens in P based on whitespaces
P^*	$\{P^i, \ldots, P^m\}$
T	set of reference strings
T_i	one string from T
$T_i^1, T_i^2, \ldots, T_i^k$	tokens in string T_i
$\|S\|$	length of string S
candidate set	strings in T that "match" current pattern P

Table 1: List of symbols used and their definitions.

Definitions. As already noted, we use P to represent the user's current query pattern. If P contains $m-1$ blocks of white space characters (for example, in the pattern "`good re`", $m = 2$) we split it into tokens $P^1, P^2 \ldots P^m$ based on the white space. We use P^* to represent this set of tokens of P. We use T to denote the *reference set*, the target strings against which P will be compared to generate the completions, where T consists of n strings over an alphabet Σ, with $T = \bigcup_{i=1}^{n} T_i$. The strings in T may also be annotated with information such as frequency, to allow sorting or selection during QAC.

For example, each T_i might be a query from a recent query log from the same search service, or might be a key-phrase that has been identified in a collection of documents by some external process, or might be a person's name if the search is to support a directory such as on a social media platform. Each string $T_i \in T$ can also be tokenized based on white spaces to get $T_i^1, T_i^2, \ldots T_i^k$.

Given these definitions, the Candidate Retrieval problem is to retrieve from the collection T a list of candidate strings (the *candidate set*) that match the query pattern P. We discuss possible semantics of the word "match" below.

By definition, Candidate Retrieval is a Boolean retrieval problem and does not involve the ranking of completions. In order to generate the final set of completions that are shown to the user, the candidates are ranked based on a scoring function, so as to identify the top-k candidates based on the choice of ranking functions.

Our contribution. We provide a taxonomy of QAC modes, explain previous literature in the field in terms of the QAC modes we describe; and survey data structures that implement these modes. We also provide preliminary experimental results that compare these alternative solutions.

2 BACKGROUND

Query auto completion (QAC) for web search was popularized by Google around 2004 [7], a feature they called *Google Suggest* at the time. The completions were generated from query logs containing past queries entered by users. Their suggestions were ordered according to the popularity of queries in the log – a method popularly known as MostPopularCompletion [2]. Several alternative methods have been proposed since then in an effort to improve the ranking of completions.

Measures based on query popularity, such as the one by Bar-Yossef and Kraus [2], assume that query popularity is static and is user-centered. More realistic approach is to consider the popularity of queries to be time-varying, both increasing and decreasing over time, with both sudden (for example, "`david bowie dead`") and gradual (for example, "`instagram`") changes in frequency. Some queries may be enduringly popular (for example, "`facebook`" as a reasonable completion for the pattern "`fa`"). Cai et al. [9] propose a time-sensitive model for QAC relying on the predicted popularity of queries. Other approaches to ranking of completions fall into: (i) learning to rank (L2R) models [8, 20], that follow supervised methods to determine a ranking; and (ii) models focusing on the information content of the completions, such as the method proposed by Cai et al. [10], which attempts to diversify the completions from the final list by removing redundant query candidates.

While these approaches seek to improve the ranking, it is our view that the Candidate Retrieval problem continues to provide a fundamental challenge, and has had relatively little attention in the literature. Most prior work defines QAC as a feature that suggests completions that contains P as a *prefix*; see, for example, Bar-Yossef and Kraus [2], Shokouhi [20], Mitra et al. [17] and Cai et al. [10]. A different querying mode is proposed by Bast and Weber [3, 5], who split P based on white space and provide matches with strings that contain all of the provided sub-patterns P^i, without regard for the order of the sub-patterns. For instance, "`gam th`" could match with "`game theory`", "`game of thrones`", "`game three`", "`the game`", and so on. Li et al. [16] provide similar functionality.

Chaudhuri and Kaushik [11] refer to the ways to match P with the string collection as "auto completion strategies", and refer to sub-string-based matching in addition to the prefix matching. Their main focus was on error-tolerant prefix match rather than the matching strategies and their implications. Ji et al. [15] describe a form of approximate QAC that allow errors in prefix matches. Nandi and Jagadish [18] extend the idea of prefix matching to phrase prediction, thereby providing completions that extend P by as many words as possible, with a trade-off in terms of the accuracy of the completion. In contrast to other approaches found in the literature, they consider only the last two words from P as the *query prefix* and generate a list of completions matching with the last two words of P. The proposed a FussyTree data structure over an alphabet of words to list the matching documents, without looking for intra-word completions. Bhatia et al. [6] consider a similar approach where they split P into two parts - the part user has finished typing and the part still being typed. A *candidate set* is then generated by intersecting the list of documents containing the completed part of the pattern with the list of documents containing possible completions for the part being typed. These completions are calculated based on prefix match.

Table 2 summarizes the previous work. The three query modes are described in the next section. The method of Nandi and Jagadish [18] doesn't fit any of the categories in the table.

There has been a significant consideration towards improving the efficiency of QAC systems. A common data structure used for generating prefix matches is a trie. In this regard, a few trie variants with different space-time trade-offs can be found in the literature, see Hsu and Ottaviano [14] and Askitis and Zobel [1]. Navarro [19] discusses a trie variant in compact setting for efficient

Citation	Mode 1	Mode 2	Mode 3
Bast and Weber [4, 5]		Implemt.	
Chaudhuri and Kaushik [11]*	Implemt.		Discussed
Ji et al. [15]*		Implemt.	
Bhatia et al. [6]		Measured	
Bar-Yossef and Kraus [2]	Implemt.		
Shokouhi [20]	Implemt.		
Mitra et al. [17]	Discussed		
Di Santo et al. [12]	Implemt.		
Cai and de Rijke [7]	Defined		
Cai et al. [10]	Implemt.		

Table 2: QAC modes described in the literature, annotated as: (1) *discussed*, the paper mentions the possibility of using that mode; (2) *defined*; the paper gives a definition for that mode; (3) *implemented*, the paper employs an implementation of that mode as part of an experimental investigation; and (4) *measured*, the paper describes performance of a mode with or without considering ranking aspects. A star (*) annotation indicates the addition of error-tolerant matching. The three QAC modes are introduced in Section 3.

auto completion along with a variants of suffix arrays and suffix trees that can be used in the QAC settings discussed in Section 3. In addition to the in-memory data structures, there are external variants of these data structures, such as Gog et al. [13]. However, in this work, we limit our scope to variants of in-memory structures.

3 QUERY AUTO COMPLETION MODES

There are several possible ways in which a complete query might be regarded as a reasonable extrapolation of a partial query. That is, there is a range of alternative ways of defining the semantics of a "match" when addressing the Candidate Retrieval problem. We call each of these possible interpretations a *mode* of QAC.

Recall that $\mathsf{T} = \mathsf{T}_1, \mathsf{T}_2, \ldots, \mathsf{T}_n$ is a set of strings against which the completions are to be generated. Each $\mathsf{T}_i \in \mathsf{T}$ can be tokenized as $\mathsf{T}_i^1, \mathsf{T}_i^2 \ldots \mathsf{T}_i^k$ based on white space. We represent the set of tokens of T_i as T_i^*. These tokens form a totally ordered set based on the order of occurrences of T_i^j in T_i^*. We use S to represent a string over the alphabet Σ, and use $|\mathsf{S}|$ to denote its length, so that it can be represented as $\mathsf{S}[1, 2, 3, \ldots |\mathsf{S}|]$. We also define a function Prefix(S) to compute the set of prefixes of S:

$$\mathsf{Prefix}(\mathsf{S}) = \{S[1 : i] \mid i \in [1, |\mathsf{S}|]\}\,.$$

With these definitions, we now describe the QAC modes.

Mode 0. (Exact Match) This is the most basic mode, and matches with only those strings in T that are an exact match with P:

$$\text{Exact Match}(\mathsf{P}) = \{\mathsf{T}_i \mid \mathsf{T}_i \in \mathsf{T} \wedge \mathsf{P} = \mathsf{T}_i\}\,. \tag{1}$$

This trivial mode is essentially binary, and returns a non-empty result only if $\mathsf{P} \in \mathsf{T}$.

Mode 1. (Prefix Match) This is probably the most common approach for matching P with the strings in T. In this mode, all strings that have P as a prefix are members of the candidate set:

$$\text{Prefix Match}(\mathsf{P}) = \{\mathsf{T}_i \mid \mathsf{T}_i \in \mathsf{T} \wedge \mathsf{P} \in \mathsf{Prefix}(\mathsf{T}_i)\}\,. \tag{2}$$

(A variant, which we do not explore further, is "Mode 1a", which completes only the current word being typed. This variant is word completion rather than query completion.)

Mode 2. (Multi-term Prefix Match) This approach splits P to obtain P^* and attempts to match queries such that, for each query token, there is at least one string token where the query token is a prefix. For example, in this mode "`ste jo`" matches with both "`steve jobs`" and "`joanne stewart`".

This scenario can be viewed as a method for selecting an initial candidate set based on the first token entered, and then iteratively filtering that set as subsequent tokens become available [3]. At each iteration, the current candidate set acts as a *context* limiting the subsequent search space. Mode 2 does not take order into account, and the matched prefixes may appear in the target string T_i in any sequence:

$$\text{Multi-term Prefix Match} = \left\{\mathsf{T}_i \mid \mathsf{T}_i \in \mathsf{T} \wedge \mathsf{P}^* \subseteq \left(\cup_k \mathsf{Prefix}(\mathsf{T}_i^k)\right)\right\}\,. \tag{3}$$

A restricted variant of Mode 2 might also be an option, requiring that the partial matches occur in the same order in T_i as they do in P (meaning that "`ste jo`" would not match "`joanne stewart`"), but we do not consider that possibility in this work.

Mode 3 (Pattern Match**).** This mode performs a standard substring match over each P^i of P. It is multi-word matching similar to Mode 2, but instead of prefix matching, a substring search is carried out over the strings T_i. Given a string S and a pattern x, we define an auxiliary function Match(S, x) as

$$\mathsf{Match}(\mathsf{S}, \mathsf{x}) = \begin{cases} \text{True}, & \text{if } \mathsf{x} = \mathsf{S}[i, \ldots i + |\mathsf{x}|] \text{ for some } i \\ \text{False}, & \text{otherwise.} \end{cases} \tag{4}$$

Pattern Match can then be defined as

$$\text{Pattern Match} = \{\mathsf{T}_i \mid \mathsf{T}_i \in \mathsf{T} \wedge (\wedge_{\mathsf{P}^k \in \mathsf{P}^*} \mathsf{Match}(\mathsf{T}_i, \mathsf{P}^k))\}\,. \tag{5}$$

Mode 4. (Relaxed Pattern Match) This mode is similar to Mode 3. A Pattern Match is performed instead of a substring match, so that answers contain the pattern within a specified edit distance or Hamming distance of the target string. As for the Match(S, x) function, we define an auxiliary function Relaxed Match(S, x, δ) that allows errors of up to δ characters:

$$\text{Relaxed Match}(\mathsf{S}, \mathsf{x}, \delta) = \begin{cases} \text{True}, & \text{if x and } \mathsf{S}[i, \ldots i + |\mathsf{x}|] \\ & \text{differ by at most } \delta \text{ for some } i \\ \text{False}, & \text{otherwise} \end{cases} \tag{6}$$

The corresponding match function is defined as:

$$\text{Relaxed Pattern Match}_\delta = \{\mathsf{T}_i \mid \mathsf{T}_i \in \mathsf{T} \wedge (\wedge_{\mathsf{P}^k \in \mathsf{P}^*} \text{Relaxed Match}(\mathsf{T}_i, \mathsf{P}^k, \delta))\} \tag{7}$$

Note that as many as δ error are permitted in each of the elemental token matches. The alternative would be to require that the total error count over the whole matching operation was limited to δ.

A further possible approach would be to make "error tolerance" an independent dimension in our taxonomy. We define it here as a separate mode for experimental and descriptive convenience, but acknowledge that it is in fact an independent dimension.

Prefix entered (P)	Mode				
	Mode 0	Mode 1	Mode 2	Mode 3	Mode 4
game of thrones	✓	✓	✓	✓	✓
game o		✓	✓	✓	✓
th gam			✓	✓	✓
gam rone				✓	✓
gam thorn					✓

Table 3: QAC matches against the target string "`game of thrones`" for the various QAC modes.

Examples. Table 3 provides examples that illustrate the differences between these modes. Note that a consequence of our definitions the strings T_i accepted as candidates by each mode are a superset of the strings accepted by lower-numbered modes. At the logical limit, one could postulate a liberal "Mode ∞" in which every string $\mathsf{T}_i \in \mathsf{T}$ was accepted as a candidate against every partial query P – hardly a useful option, of course. That is, for the majority of applications the preferred mode will not be the one that results in the most candidates being generated, it will be the one that provides the most useful candidate set, where "useful" may be subjective, and ultimately based on user reaction and preference.

4 SYNTHETIC PATTERN GENERATION

A challenge that arises when evaluating the performance of any QAC system is to obtain a suitable sequence of queries. Detailed query logs from large search services contain the type of interaction data required for this purpose, but, because of privacy concerns, and the difficulty of adequately anonymizing such data, they are rarely disclosed. To effectively measure QAC performance, the query sequence should record each user keystroke, the suggestions that were presented to them with each of their actions, any word or phrase selection actions that accelerated query formulation, and the final query that they generated.

In the absence of user interaction data, a common approach is to assume possible ways in which the final pattern P is constructed, and submit these to the QAC system. For example, Bast and Weber [3] submit the queries from a simple query log one character at a time with a minimum length of three, while Bhatia et al. [6] evaluated their model by submitting the first (key)word from the queries to the QAC system. In the Bast and Weber approach, if the final query pattern submitted is "`weather today`", the QAC system attempts to find suggestions for each keystrokes. For this query pattern, the partial queries are (possibly subject to some lower bound on length) "`w`", "`we`", "`wea`", and so on.

However, users interact with QAC systems in richer ways than just appending one character at a time to a growing query. A "step" in an interaction might be: typing a character; choosing one of the completions the system has presented; typing backspace to delete a character (either previously typed, or from the end of a chosen completion); or just submitting a string without bothering to complete typing it, hoping that the search engine would still give them the expected results via the in-built "assuming that you meant" functionality, despite the missing information.

We now formalize those observations, and describe a generative model for synthesizing query patterns P as a progression through a state machine, thereby allowing plausible QAC sequences to be generated for the purposes of evaluation. In this regard, we build on the work of Chaudhuri and Kaushik [11], who describe the actions a user can perform when operating in a QAC environment.

We start by choosing a string $\mathsf{T}_i \in \mathsf{T}$ (recall that T is the collection of strings from which selections are to be made), based on which we generate the query patterns. If we set $\mathsf{P} = \epsilon$, the empty string, we can define a set of possible interactions as follows:

(1) *Append*: Append a character c to the pattern. To account for typing errors, we model this operation in such a way that, based on a probability distribution, the user either enters the next character from T_i or makes a typing error and enters one of the adjacent keys from a standard keyboard layout.
(2) *Jump*: Navigate through the top-k completions presented using the mouse of the arrow keys on the keyboard, and choose one of the completions.
(3) *Delete*: Remove (backspace) one character from P, provided $|\mathsf{P}| \geq 1$. In principle, the user could remove a set of characters from any location in P after using the mouse to navigate within the pattern. However, we make a simplifying assumption here that appends and deletions operate on a *stack* model.
(4) *Submit*: Hand the current query P to the underlying search system and terminate the interaction with the QAC interface. No further suggestions are displayed after this action.

The intention is that string T_i be used to generate a sequence of patterns that might be entered by a user who wishes to pose T_i as a query. The use of a stack model means that this mechanism is not ideal for testing QAC Mode 3; moreover, we have neither data nor intuition that would support any specific artificial model of generating appropriate substrings.

With the structure of use interactions explained, we now define two models for generating synthetic query patterns, which we can then use to evaluate the performance of QAC systems. We rely on a probabilistic automaton, in which the transitions are the user interactions defined earlier and the states in the automaton define the different states that a QAC system is in when the interactions takes place. Both models make use of these states:

(1) q_0: Initial state. The QAC system is in this state when the user has an information need and decides to initiate a search.
(2) q_m: Modification state. The partial query gets modified in this state. The system will be in this state while the user modifies their query by a series of *Append* or *Delete* operations.
(3) q_c: Choose a completion state. When the user scrolls through the suggestions using the down arrow key or mouse (that is, performs a *Jump* operation, the system moves to this state.
(4) q_s: This is the final state. The user submits their final query pattern to the underlying search service and terminates their interaction with the QAC system.

Interaction with the QAC system starts when the user begins typing their first character $c \in \Sigma$ in the search box. In the models explained in the following sections, this initial transition is assumed to be implicit to the system.

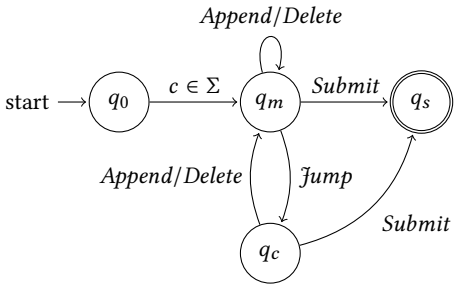


Figure 1: State diagram for the *LR* and *GEN* models. In the former, the probabilities associated with the *Jump* and *Delete* transitions from q_m are both zero, and pattern prefixes are generated strictly left to right.

***LR* model.** In this simple automaton partial strings are generated by appending characters from the seed string T_i in left to right order, using fixed probabilities [3]. As long as there are further characters remaining to be appended, the machine continues adding them. It then submits the final string and terminates. That is, given a target string $\mathsf{T}_i \in \mathsf{T}$, *LR* appends each character $c \in \mathsf{T}_i$ to obtain the partial query patterns, P[0], P[0..1], P[0..2], and so on, leading to a submitted pattern P.

***GEN* model.** This is a more general probabilistic automaton that allows two additional actions, *Jump* and *Delete*. The shifts between the states occur with some given probability that is part of the parameterization of the process. Figure 1 shows the state transitions for the *GEN* model, and highlights the difference between it and the *LR* approach – the two proposed models differ from each other only in regard to the range of transition probabilities associated with state q_m.

For a seed string T_i, this model generates a sequence of possible patterns (starting with a single character) that might have been entered by the user in the search box before submitting T_i. In contrast to *LR*, this model may generate patterns that are not substrings of T_i in two situations: (i) if there is a *Append* operation with typing error; or (ii) if a transition to *Jump* occurs, in which case, further actions are performed on the completion string chosen. The model terminates the generation process (performs a *Submit* operation) whenever the generated pattern is same as the seed string T_i.

The state transition probabilities can be static or dynamic, and also conditioned on the length of the prefix being generated, or any of a range of other factors. For example, given a character-by-character detailed query log, these probabilities might be learned, to generate synthetic test data that mimics user interactions with that or a similar QAC system. In the absence of a log, in this preliminary work, we assume static transition probabilities.

5 EXPERIMENTS

We have described a range of querying modes, and explained how they affect the completions that are shown to users. This section summarizes the data and implementations we have explored, and then measures the various modes via two pertinent characteristics: the frequency-based rank of the sought query string; and the computation time using a range of data structures. Our aim is not to provide an exhaustive examination, but to demonstrate how our taxonomy can be used effectively as the basis of comparison of QAC systems. We acknowledge the limitation of our artificial queries as a tool for measuring Mode 3, and as yet have not completed an implementation of Mode 4.

Data resources. We use a query log obtained from a site-specific search engine associated with an Australian University as a source for target strings to be used in the synthesis process, and as the basis for extraction of candidate sets. The queries from the collection are mapped to Unicode using utf-8 encoding. An implication of this encoding is that our alphabet set Σ can extend beyond the ASCII character set and provide support for additional Unicode characters. Non-alphanumeric Unicode characters as well as English stop words are removed from the strings in the log. All strings are mapped to lowercase letters to further reduce the size of Σ. After the pre-processing steps are applied, there are 72,245 unique queries in the query log, over an alphabet of $\Sigma = 71$ unique characters. The total length of the unique strings $\sum_i T_i = 1{,}102{,}031$, with a mean length of 15.25. We sample 1% of the strings from this set to build a set of QAC test strings, each of which is to applied against the full set. That is, the test set of QAC queries is based on 722 randomly selected seed strings, and has a total length of 10,774 bytes.

The underlying implementations are written in C/C++. We use Python wrappers of these libraries in our experiments. The reported time and memory are measured by running the code on a Macbook Pro with 2 GHz Intel Core i5 and 8 GiB 1867 MHz LPDDR3 RAM and 256 GiB PCIe-based on-board SSD.

Mode 1 implementation. A natural choice for implementing Mode 1 is a trie data structure. A trie stores each string T_i as a root-to-leaf path. The branches of the trie are labeled with characters from Σ. Tries are known to provide fast prefix look up, with one key parameter determining their performance being the choice of data structure used to represent the nodes. In our experiments, we measure the performance of Mode 1 relying on three open source trie implementations – Marisa trie[1]; DAWG trie[2]; and DATrie[3].

Mode 2 implementation. Bast and Weber [5] provide a well-known implementation of this mode. In their investigation, they compare inverted indexes with the proposed hybrid data structure (HYB) and show that HYB achieves faster querying time with approximately the same volume of memory as an inverted index. They also report a range of performance benchmarks.

In the initial version of our implementation, we tokenize each query from the log using white space, and use an inverted index (implemented using Python dictionary) to create a mapping from words to the strings they appear in. A trie built on the set of tokens is used to obtain the tokens matching a given prefix. Using the set of words, a list of queries these tokens appear in are retrieved from the inverted index. The subsequent processing follows the description of Bast and Weber.

Mode 3 implementation. As far as we know, there are no existing public domain implementations that can be used for generating Mode 3 completions. General pattern search has a long history of

[1] https://github.com/pytries/marisa-trie
[2] https://github.com/pytries/DAWG
[3] https://github.com/pytries/datrie

String $T_i \in T$	Patterns generated from T_i
hotel design	h, ho, hot, hote, hotel, hotel, hotel d, hotel de, hotel des, …, hotel desig, hotel design
abbreviations	a, ab, abb, abbr, abbre, abbrev, abbrevi, abbrevia, abbreviat, abbreviati, …, abbreviations

Table 4: Example QAC patterns generated by the *LR* model.

String $T_i \in T$	Patterns generated from T_i
outdoors club	o, ou, our, ou, out, …, outdoora, outdoor, outdoors, …, outdoors clu, outdoors club
abbreviations	a, av, a, ab, abb, abbt, abb, abbr, abbreviations

Table 5: Example QAC patterns generated by the *GEN* model.

algorithmic development. In the general case, when searching a target string of length n for instances of a pattern of length m, two different approaches are possible. Non-indexed search mechanisms (and no preprocessing of the target string) such as the KMP and Boyer-Moore-based approaches, require time linear in n, assuming that $m \ll n$. The best known approaches to indexed pattern search (using a pre-computed structure that occupies additional space, with the cost of index construction assumed to be amortized to zero by applying it without alteration to an arbitrary number of search strings) involve suffix tree or suffix array structures, and require $O(m + occ)$ time, where occ is the number of times that the pattern appears in the string.

Here we evaluate the performance of Mode 3 Candidate Retrieval problem using both of these options. In the first, we use Python stringlib fastsearch[4] algorithm, which is a mix of Boyer-Moore and Horspool methods, and can be expected to work faster than KMP. In the second approach, we we use a generalized suffix tree for storing the strings T_i that are the target matches. That is, we use a suffix tree for identifying the strings that match a given pattern and use an inverted index for intersecting these lists. The suffix tree implementation is taken from the Strmat[5] library.

Synthetic query pattern generation. We evaluate the performance of Mode 1, Mode 2, and Mode 3 implementations on query patterns generated using both *LR* and *GEN* models (Section 4). Generating *LR* query patterns is straightforward and does not require transition probabilities, and examples are shown in Table 4.

To generate *GEN* patterns, in this initial study we assume static transition probabilities, and adjust them to generate plausible query patterns. We set the initial transition probability from any state to the *Append* state as 0.80, the probability of transition from any state to *Delete* as 0.04, and *Jump* probability as 0.16 throughout. These values were chosen as plausible ones after exploration of a range of options.

To simulate typing errors in *Append* operations, we further assume a Gaussian distribution over characters in the "within two" neighborhood of the next target character in T_i, based on a standard QWERTY keyboard layout. The mean of the Gaussian is set as the next character in T_i, so that it gets chosen in most cases with a standard deviation of 0.19 (also chosen empirically). If that Gaussian distribution does give rise to a "typing error", then we set the next operation as a *Delete* with a fixed probability of 1.0.

The *Jump* operation picks a matching completion from amongst the top-10 completions based on the QAC modes when sorted by the frequency of the string in the set T. To be eligible to be selected, a completion T_i must have the current query string P as a prefix. We keep this assumption throughout all three modes. In reality, a user entering patterns in a Mode 3 supported system would choose a completion that has the tokens from the pattern as a substring, a situation that is more complex than we wish to consider at this time. If the completion selected by that process is longer than T_i, then we perform *Delete* operations until the pattern reduced to a length at which T_i is again a valid candidate match.

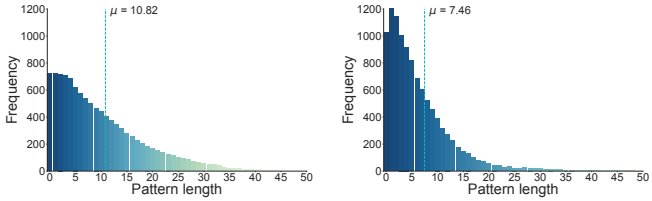
Finally, when the generated pattern is equal to T_i, we do a *Submit* and terminate generating patterns for that document. Sample query patterns obtained from *GEN* following this process are given in Table 5.

We again note that the transition probabilities should ideally be conditioned on the pattern length, and that the probability of submit a pattern P should increase with its length. Such an approach would ensure that the generated query patterns are a good match with what is encountered in practice. Since the completions displayed vary with the QAC modes, the patterns generated by *GEN* for Mode 1, 2 and 3 will be different. Our aim is to compare the changes in runtime of the QAC systems when supplied with query patterns generated from *LR* and *GEN* models. In order to achieve a fair comparison, we make sure that the same number of query patterns generated by the two models are submitted to the QAC system during evaluation. Figure 2 shows the distribution of prefix lengths obtained by generating query patterns using *GEN* for the modes.

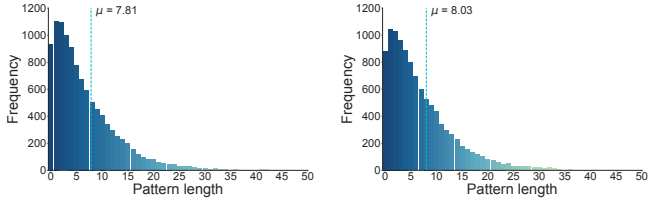
Patterns generated by the *LR* model will always have additional characters added to a pattern until it is submitted. With this simplification, we can greatly improve the performance if our data structure is state preserving. Considering Mode 1, if we retain a finger to the last trie node visited, then when the model appends the next character, we can resume traversal from that node to generate further completions. It is only when the user submits a pattern that we start traversing from the root of the trie again. This assumption does not hold for the patterns obtained from *GEN* and there could be multiple resets before the user submits the pattern. Each time they choose one of the completions, the search for a new match need to start from the root of the trie and additional mechanisms are needed to deal with *Delete* operations. In this work, we do not investigate the effect of persistence and leave measurement of those performance gains for future work.

A key factor determining the querying time is the length of pattern P. When the user enters the first character, there will be a large number of possible matches, and listing all of them is time-consuming. If we use *LR* to generate the patterns, their length distribution will be monotonically decreasing as shown in Figure 2a.

[4] https://hg.python.org/cpython/file/a206f952668e/Objects/stringlib/fastsearch.h
[5] https://github.com/JDonner/SuffixTree



(a) *LR* (common to all three modes) **(b)** *GEN* on Mode 1 completions



(c) *GEN* on Mode 2 completions **(d)** *GEN* on Mode 3 completions

Figure 2: Distribution of QAC query lengths obtained using *LR* and *GEN* for completions generated based on Mode 1, Mode 2 and Mode 3. In Mode 3, with the transition probabilities selected, the most common QAC length is three.

Patterns generated from *GEN* will have a different length distribution compared to the patterns from *LR* depending on the transition probabilities. Using the transition probabilities we set in our experiments, we get the distribution of pattern lengths for Mode 1, Mode 2 and Mode 3 as shown in Figures 2b, 2c, and 2d.

Querying time. To measure the querying time, we generate query patterns using *LR* and *GEN* models from the documents in our test collection. These pre-computed patterns are then fed into our QAC modes implementations to list the documents matching with the patterns. Querying time is purely the time taken to list the documents and doesn't involve the cost of sorting them to generate top-k completions. Recall that we are focusing on Candidate Retrieval, and do not account for the time needed to sort the completions.

In our experiments, we used Python wrappers of the implementations and the reported time is measured using Python's *timeit* module. Making calls to C/C++ libraries from Python would introduce some latency. However, since we are measuring the time using Python wrappers for all the implementations, we believe that the difference is negligible and the results are comparable.

Times are shown in Table 6, and demonstrate that *GEN* queries take longer to execute than *LR* queries, presumably as a consequence of them on average being shorter (because of the backspace operation). The Mode 3 queries are very slow to execute, and (at least at the scale of the current experimentation) exhaustive string search is only a factor of two slower than the suffix tree approach.

Index construction time. In addition to the querying time, we measure the index construction time for each implementations. Depending on the nature of the system, it is important to take this time into account. The implementations used here are static and do not support dynamic update or delete operations. Building time is defined as the time needed to construct the index structure after all the documents have been loaded into the memory and all the required pre-processing operations are performed. It is also worth mentioning here that all our indexes are in memory and no disk operation costs whatsoever are involved in the evaluation process.

Index memory. In our experiments, we measure the increase in memory before and after the index is loaded into the memory. The index memory reported in Table 6 are the increase in memory as reported by the Python memory profiler package.[6]

As can be seen in Table 6, there is a very wide range of space needs for the various structures. The largest structure, the suffix tree plus inverted index arrangement, required between five and ten times the space of the tries used for Mode 1 operations.

Variation in ranking with modes. One approach for evaluating the ranking mechanism of QAC systems is to generate the list of ranked completions on a set of query patterns and use a standard evaluation metric like reciprocal rank (RR) to get an estimation of the goodness of ranking. Existing work on QAC adapts one of the QAC modes and reports variations in the ranking. We show that as consequence of having more than one way to match the pattern with the documents, there is substantial variation in the final rankings of the completions across the QAC modes.

Since matching acts like a filter on the document collection, different QAC modes returns a different set of candidates for the same pattern P. For each document T_i in our test collection (1% documents from the collection), we generate patterns using *LR* and obtain matching candidates for each QAC modes. The candidates are then ranked by their frequency and the rank of T_i among this ranked list is computed.

Figure 3 shows the variation in median rank of the documents in the list of completions with the length of the pattern for patterns generated by the *LR* model. We do not consider patterns from *GEN* in this experiment because whenever a generated pattern is not a substring of the seed string (as explained in Section 4) the seed string may not be in the list of completions. This leads to a lot of *no-hit* cases and the variation in rank is irregular. From the figure, there is significant variation in ranking for the same set of patterns across the three QAC modes. Since most of the previous investigations are based on only one of the QAC modes, and on simple left-to-right keystroke models, these results show that previous conclusions are dependent on assumptions that may not be realistic.

6 CONCLUSIONS

We have considered issues related to query auto completion systems – matching a partially-typed pattern against a set of target strings to generate a *candidate set*. We call this the Candidate Retrieval problem, and identify a range of modes that might be of interest. Our results shows that for the same ranking scheme, the order in which the completions are presented can vary significantly across the QAC modes.

We also proposed a pattern generation model for evaluating the performance of QAC systems. The *GEN* model can capture realistic user interaction and generate query patterns with a range of length distributions when compared to models that generates patterns

[6] https://pypi.python.org/pypi/memory_profiler

Mode	Data structure	Build time (millisec)	Query time (ms)		Index memory (MiB)
			LR	*GEN*	
Mode 1	Marisa trie	106	2160	5220	12.9
	DAWG Trie	221	1530	4370	4.27
	DATrie	363	8720	14,200	1.46
Mode 2	Marisa trie + inverted index	140	16,500	24,100	10.2
	DAWG Trie + inverted index	170	15,600	23,700	8.55
	DATrie + inverted index	436	22,000	29,700	8.02
Mode 3	Suffix Tree + inverted index	662	906,000	981,000	53.9
	Python substring search	—	2,090,000	1,940,000	—

Table 6: Execution time and memory cost of QAC modes using 722 distinct seed strings then tested against a collection of 72,245 strings. A total of 10,774 patterns are evaluated, see the text for details. Each of the times reported here is the total time to process all 10,774 queries, with the minimum over three independent runs taken.

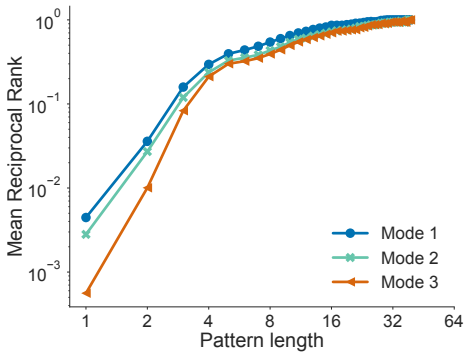


Figure 3: Variation in candidate ranks with length of the pattern and QAC modes, for LR query set.

from left to right based on a seed string. Our model also captures the effect of typing errors, jumps from one pattern to another, and character deletions.

Relying on available implementations of the underlying data structures, we provide an initial framework for measuring QAC modes. The performance of these implementations is evaluated in terms of index memory, index construction time, and querying time. Our experiments show that the variation in pattern length distribution from these two models result in notably different querying time for the test patterns.

We plan to build on this preliminary investigation by undertaking a detailed analysis of the proposed *GEN* model. Further, the transition probabilities need to be learned from a query log that captures the actions from the users, or by other methods. The effect of persistence and state (keeping track of the last location in the index structure) between patterns generated by the two models is another interesting direction that waits to be explored.

Acknowledgment. Unni Krishnan receives top-up scholarship from the Microsoft Research Centre for Social Natural User Interfaces (SocialNUI) at The University of Melbourne; and his participation in the conference was funded by an ADCS travel grant.

REFERENCES

[1] N. Askitis and J. Zobel. Redesigning the string hash table, burst trie, and BST to exploit cache. *ACM J. Exp. Alg.*, 15:1–7, 2010.

[2] Z. Bar-Yossef and N. Kraus. Context-sensitive query auto-completion. In *Proc. WWW*, pages 107–116, 2011.

[3] H. Bast and I. Weber. Type less, find more: Fast autocompletion search with a succinct index. In *Proc. SIGIR*, pages 364–371, 2006.

[4] H. Bast and I. Weber. When you're lost for words: Faceted search with autocompletion. In *Proc. SIGIR Wrkshp. Faceted Search*, pages 31–35, 2006.

[5] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR & DB integration. In *Proc. CIDR*, pages 88–95, 2007.

[6] S. Bhatia, D. Majumdar, and P. Mitra. Query suggestions in the absence of query logs. In *Proc. SIGIR*, pages 795–804, 2011.

[7] F. Cai and M. de Rijke. A survey of query auto completion in information retrieval. *Found. & Trends in Inf. Ret.*, 10:273–363, 2016.

[8] F. Cai and M. de Rijke. Learning from homologous queries and semantically related terms for query auto completion. *Inf. Proc. & Man.*, 52(4):628–643, 2016.

[9] F. Cai, S. Liang, and M. de Rijke. Time-sensitive personalized query auto-completion. In *Proc. CIKM*, pages 1599–1608, 2014.

[10] F. Cai, R. Reinanda, and M. de Rijke. Diversifying query auto-completion. *ACM Trans. Inf. Sys.*, 34(4):25:1–25:33, 2016.

[11] S. Chaudhuri and R. Kaushik. Extending autocompletion to tolerate errors. In *Proc. SIGMOD*, pages 707–718, 2009.

[12] G. Di Santo, R. McCreadie, C. Macdonald, and I. Ounis. Comparing approaches for query autocompletion. In *Proc. SIGIR*, pages 775–778, 2015.

[13] S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth. Large-scale pattern search using reduced-space on-disk suffix arrays. *IEEE. Trans. Know. Data Eng.*, 26(8):1918–1931, 2014.

[14] B.-J. P. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proc. WWW*, pages 583–594, 2013.

[15] S. Ji, G. Li, C. Li, and J. Feng. Efficient interactive fuzzy keyword search. In *Proc. WWW*, pages 371–380, 2009.

[16] G. Li, S. Ji, C. Li, and J. Feng. Efficient type-ahead search on relational data: a tastier approach. In *Proc. SIGMOD*, pages 695–706, 2009.

[17] B. Mitra, M. Shokouhi, F. Radlinski, and K. Hofmann. On user interactions with query auto-completion. In *Proc. SIGIR*, pages 1055–1058, 2014.

[18] A. Nandi and H. Jagadish. Effective phrase prediction. In *Proc. VLDB*, pages 219–230, 2007.

[19] G. Navarro. *Compact Data Structures: A Practical Approach.* Cambridge University Press, 2016.

[20] M. Shokouhi. Learning to personalize query auto-completion. In *Proc. SIGIR*, pages 103–112, 2013.