Compressed Pattern Matching in the Annotated Streaming Model

Markus Jalsenius\textsuperscript{1}, Benjamin Sach\textsuperscript{1}, and Søren Vind\textsuperscript{2}\textsuperscript{*}

\textsuperscript{1} University of Bristol, Department of Computer Science
\textsuperscript{2} Technical University of Denmark, DTU Compute

Abstract. The \textit{annotated streaming model} was originally introduced by Chakrabarti, Cormode and McGregor [ICALP 09]. In this extension of the conventional streaming model, a single query on the stream has to be answered by a \textit{client} with the help from an untrusted \textit{annotator} that provides an annotated data stream (to be used with the input stream). Only one-way communication is allowed. We extend the model by considering multiple queries, and focus on on-going queries where a new output must be given every time an input item arrives.

In this model, we show the existence of a data structure that enables us to store and recover information about past items in the stream using very little space on the client. We first use this technique to give a space-annotation trade-off for the annotated multi-indexing problem, which is a natural generalisation of the previously studied annotated indexing problem.

Our main result is a space-annotation trade-off for the classic exact pattern matching problem in phrase-compressed strings. In particular, we show the existence of a $O(\log n)$ time per phrase, $O(\log n + m)$ client space solution which uses $O(\log n)$ words of annotation per phrase. If the client space is increased to $O(n^\epsilon + m)$ then $O(1)$ words of annotation and $O(1)$ time per phrase suffices. Here $n$ is the length of the stream and $m$ is the length of the pattern. Our result also holds for the well-known LZ78 compression scheme which is a special case of phrase-compression.

All of the problems we consider have $\Omega(n)$ randomised space lower bounds in the standard (unannotated) streaming model.

1 Introduction

In the \textit{streaming model} \cite{1,10,21}, an \textit{input stream} of elements arrive one at a time, and we must solve problems using sub-linear (typically polylogarithmic) space and time per element with a single pass over the data. Throughout the paper, we let $n$ denote the length of the input stream, and assume that input elements require $O(1)$ words of $w \geq \log n$ bits each.

In order to model the current state of affairs in computing with easy and cheap access to massive computational power over the internet, the \textit{annotated streaming model} introduced by Chakrabarti, Cormode and McGregor \cite{3,4} expands the normal streaming model by introducing an untrustworthy \textit{annotator}. This annotator is assumed to have infinite computational resources, and it assists a \textit{client} in solving some problem by providing an \textit{annotated data stream} that is transmitted along with the normal input stream (i.e. the client-annotator communication is one-way). Software or hardware faults,

\textsuperscript{*} Supported by a grant from the Danish National Advanced Technology Foundation.
as well as intentional attempts at deceit by the annotator, are modeled by assuming that
the annotator cannot be trusted. Consequently, for a given problem we must create a
client algorithm and associated annotation protocol that allows the client to either solve
the problem if the annotator is honest, or alternatively to detect a protocol inconsistency.
As we are designing an algorithm-protocol pair, we allow the annotator to know the
online algorithm used by the client (but, crucially, not the random choices made by
the client at runtime). This means that the annotator can simulate the client and its
behaviour, up to random choices.

In this paper we introduce a variant of the annotated streaming model which is
suited to on-going, high-throughput streaming pattern matching problems. In particular
our main result in this paper is a randomized annotated streaming algorithm which
detects occurrences of a pattern in a compressed text stream as they occur with high
probability. In contrast to the standard annotated streaming model, our result returns
an answer (to the implicit query ‘is there a new match?’) every time a stream element
arrives. We also provide worst case bounds on the amount of annotation received by
the client between any two text stream elements. This is important in high-throughput
applications. A particularly notable feature of our annotated streaming algorithm is
that in uses $o(n)$ space which is impossible (even randomized) for this problem in the
standard unannotated streaming model. In fact at one point on the space-annotation
trade-off that we present we use only $O(\log n + m)$ space (where $m$ is the pattern length)
while still maintaining $O(\log n)$ worst case time and annotation per stream element. Our
result also holds for the widely used LZ78 compression scheme which is a special case of
phrase-compression.

Annotated Data Structures. We also introduce the notion of an annotated data structure
that is run by the client and the annotator in cooperation. The annotator “answers
queries” in the annotated data stream, and the client maintains a small data structure for
checking the validity of annotated query answers. Answers can be given by the annotator
if the queries are specified only by the client algorithm and the input, meaning that the
annotator can predict required answers even though no two-way communication takes
place.

The annotated data structure we present relies on using Karp-Rabin fingerprints
on the client to ensure that the annotator remains honest. As a result, all answers are
correct with high probability. As the annotator has unbounded computational power, it
is crucial that they do not have access to the random choices of the client.

1.1 The model

Before we give our results, we give a more detailed overview of the key features of the
variant of the annotated streaming model that we consider. In each case we highlight
how this compares to the standard annotated streaming model.

Multiple Queries. The standard annotated streaming model supports a single query -
which occurs after the full input stream arrives. For streaming pattern matching prob-
lems, it is conventional and natural to require that occurrences of a pattern in the text
are reported as they happen. That is, we consider each new element in the stream as a new query that must be answered before the next stream element arrival. We extend the model by allowing multiple queries, where queries may be interleaved with the input stream. This extension is natural and necessary for streaming pattern matching problems where matches must be found as they occur. In our model this corresponds to performing a query after every stream element arrives.

Annotations. In both the standard annotated streaming model and our variant, the annotations can be modelled as additional words which arrive between consecutive elements of the input stream. In the standard model, annotation is measured by the total number of words of annotation received by the client while processing the entire stream. In contrast we will give bounds on the worst-case annotation per element in the input stream. I.e. the maximum number of words of annotation received by the client between any two input elements. These annotation per element guarantees are important in high-throughput applications where queries are highly time sensitive as may well be the case for pattern matching problems. It is also important in applications where the arrival rate of the original stream cannot be controlled. There may simply not be time to receive many words of annotation before the next stream element arrives.

Prescience. In our variant of the annotated streaming model, we assume that the annotator is also prescient (as in parts of [3,4]), meaning that it has full access to the input stream in advance. In the case that the annotator also provides the input stream this is a very reasonable assumption, and the model we will use in this paper. Our motivation is that the client may require an annotated stream when receiving some input stream to be able to verify if the input stream is valid. For example, our scheme allows the client to perform pattern matching in a compressed input text. That is, the client can for example perform a streamed virus scan (using virus signatures as patterns) on a streamed input file. The result is that a malicious prescient annotator trying to infect the client can not make the client receive an infected file.

1.2 Our Results

As our main result, we solve the pattern matching problem in a phrase-compressed text stream where phrases arrive in order, one by one. The compression model is classic, with each phrase being either a single character or an extension of a previous phrase by a single character. This model subsumes for example the widely used LZ78 compression scheme.

This is the first result to show the power of one-way annotation in solving classic problems on strings, proving that the annotator allows us to reduce the space required by the client from linear in the stream length to logarithmic by using a logarithmic amount of annotation per phrase received. We give the following smooth trade-off for the problem:

**Theorem 1.** Let \( 2 \leq B \leq n \). Given a text compressed into \( n \) phrases arriving in a stream and a pattern \( p \) of length \( m \). We can maintain a structure in \( O(B \log_B n + m) \)
space that allow us to determine if an occurrence of \( p \) ends in a newly arrived phrase in \( O(\log_B n) \) words of annotation and \( O(\log_B n) \) time per phrase. Our algorithm is randomised and all matches are output correctly with high probability (at least \( 1 - 1/n \)).

That is, we can solve the problem in \( O(\log n + m) \) space and \( O(\log n) \) time and annotation per phrase; or if spending \( O(n^\epsilon + m) \) space then \( O(1) \) time and annotation per phrase suffices. In the standard streaming model, the problem has a \( \Omega(n) \) randomised space lower bound, which show that one-way communication from an untrusted annotator can help in solving classic string problems.

The result is a careful application of techniques for pattern matching in compressed strings, providing a simple initial solution in linear space and constant time per phrase with no annotation. We reduce the space required using the new annotated data structure described below that allows us to store and access arbitrary information in logarithmic client space with logarithmic overhead.

Before giving the solution to Theorem 1, we give an interesting warm up with a solution to streamed multi-indexing, motivating and illustrating our solution scheme. From a high level, we solve the problems in three steps:

1. Construct a protocol for the annotation that must be sent by the annotator when an element arrives in the stream.
2. Give a client algorithm that uses the annotation to either solve the problem or to detect a protocol inconsistency before the next input arrives.
3. Store information about the current element for the future, and consistently retrieve required information about the past.

We show the existence of the following new data structure for the streamed recovery problem that generally allows us to trade client space for annotation when storing information about the stream in step 3. We believe this data structure to be of independent interest. We associate with each stream input item an automatically incremented timestamp \( t \), and the problem is to maintain an array \( R \) with an entry for each timestamp. The operations are attach \((i, x)\), which sets \( R[i] = x \) (i.e. it modifies a given array element); and recover() returns the data \( R[t] \) associated with the current timestamp \( t \).

We show the following theorem:

**Theorem 2.** Let \( 2 \leq B \leq n \). There is an annotated data structure for streamed recovery that requires \( O(B \log_B |R|) \) words of space, and executes operations in \( O(\log_B |R|) \) words of worst case annotation and \( O(\log_B |R|) \) time. The result is randomized and all operations are completed correctly with high probability (at least \( 1 - 1/n \)).

### 1.3 Related Work

Chakrabarti, Cormode and McGregor \[3,4\] introduced the annotated streaming model and gave solutions to a number of natural problems, showing the utility of the annotator in solving classic problems on streams, such as selecting the median element, calculating frequency moments and various graph problems. Assuming a helpful annotator, Cormode, Mitzenmacher and Thaler \[9\] show how to solve various graph problems such as
connectivity, triangle-freeness, DAG determination, matchings and shortest paths. Semi-streaming solutions requiring superlinear space and annotation to solve triangle counting and computing maximal matchings was given by Thaler [24].

Multiple papers [3,8,19,20] have considered a variant allowing interactive communication (where the client can query the annotator in a number of rounds). Chakrabarti et al. [5] showed that very little two-way communication is sufficient to solve problems such as nearest neighbour search, range counting and pattern matching, restricting the amount of interaction to be only a constant number of rounds. Klauck and Prakash [19] consider two-way communication, but similarly to our model variant restrict the amount of annotation spent per input element, giving solutions to the longest increasing sub-sequence problem (and others). To the best of our knowledge, there are no proposed solutions to any classic string problems where only one-way communication is allowed.

The annotated streaming model is related to a myriad of models from other fields where an untrusted annotator helps a client in solving problems (see e.g. [2,13–16]). For example, in communication complexity an Arthur-Merlin Protocol [2,15] model an all-powerful but untrustworthy Merlin that help Arthur to solve some problem probabilistically (using only public randomness). In cryptology, a related notion is that of Zero Knowledge Proofs [13,14], where a client must verify a proof by the annotator without obtaining any knowledge about the actual proof (here, private randomness is permitted).

**Karp-Rabin fingerprints.** Our work makes use of Karp and Rabin [18] fingerprints which were originally used to design a randomized string matching algorithm and since have been used as a central tool to design algorithms for a wide range of problems (see e.g., [7,17,22]). The Karp-Rabin fingerprint of a string is given by the following definition.

**Definition 1.** Karp-Rabin fingerprint. Let $p$ be a prime and let $r$ be a random integer in $\{1, 2, 3, \ldots, p-1\}$. The fingerprint function $\phi$ for a string $S$ is given by:

$$\phi(S) = \sum_{i=0}^{\lfloor |S|/2 \rfloor} S[i]r^i \mod p.$$  

We will make extensive use of the following well-known properties. Given $\phi(S)$ and $\phi(S')$, we can compute the fingerprint of the concatenation $\phi(S \circ S')$ in $O(1)$ time. Given $\phi(S)$ and $\phi(S \circ S')$, we can compute the fingerprint $\phi(S')$ in $O(1)$ time. If $p > n^4$ then $\phi(S) = \phi(S')$ iff $S = S'$ with probability at least $1 - 1/n^3$. This is the only source of randomness in our results. For convenience in our algorithm descriptions and correctness we will assume that whenever a comparison between some $\phi(S)$ and $\phi(S')$ is made that $\phi(S) = \phi(S')$ iff $S = S'$. As our client algorithms run in sub-quadratic total time, by applying the union bound, we have that this assumption holds for all comparisons with probability at least $1 - 1/n$ when $p \geq n^4$.

## 2 Multi-Indexing

As an interesting warm-up before showing our main results, we show how to use Theorem 2 to solve the multi-indexing problem. The input stream consist of a sequence of input
elements \( X \) and queries \( Q \). The answer to an \texttt{index}(i) query is the input element \( X[i] \) (where \( i \) is an index in the input element sequence). Input elements and queries may be mixed (but queries must refer to the past). At any time \( n = |X| + |Q| \) is the stream length so far.

In the standard streaming model (without annotation) it can be shown that even a randomized algorithm must use \( \Omega(|X|) \) bits of space on the client. This follows almost immediately by the observation that a streaming algorithm for multi-indexing which uses \( o(|X|) \) bits of space would give an \( o(|X|) \) bit one-way communication protocol for the indexing problem. It is folklore that this is impossible. In the annotated streaming model without prescience, Chakrabarti et al. [3] gave a lower bound in the form of a space-annotation product of \( \Omega(|X|) \) bits if \( |Q| = 1 \) (and an upper bound with \( O(\sqrt{|X|}) \) space and annotation). There are no better lower bounds for \( |Q| > 1 \) or when having access to prescience.

There are two simple solutions, one of which is using \( O(1) \) space to store a Karp-Rabin Fingerprint of \( X \). To answer an \texttt{index}(i) query, the annotator must then replay all of \( X \) in \( O(|X|) \) annotation, which allows the client to answer the query and verify that \( X \) was correctly replayed.

If the client instead stores the entire stream in \( O(|X|) \) space, it is easy to answer a query in \( O(1) \) time, by simply looking up the answer. This is the solution we will build on, using the data structure of Theorem 2 as black box storage of \( X \) to reduce the space use, resulting in the following trade-off:

\textbf{Theorem 3.} Let \( 2 \leq B \leq n \). We can solve the multi-indexing problem in \( O(B \log_B n) \) space, using \( O(\log_B n) \) words of annotation and \( O(\log_B n) \) time per stream element. All queries are answered correctly with high probability (at least \( 1 - 1/n \)).

Clearly, we can answer a query \texttt{index}(i) if we have access to element \( X[i] \). Assume that on arrival of element \( X[i] \) we know that at time \( t \) query \texttt{index}(i) will arrive. This allows us to perform an \texttt{attach} operation for \( X[i] \) at time \( t \), and we can then use a \texttt{recover} query to get the element. If we have these timestamps for all queries, we clearly have \( |R| = O(n) \) when using Theorem 2. As the annotator is prescient, it has all the timestamps for queries to \( X[i] \) and can send them to the client when the element arrives in the stream. The resulting annotation takes \( O(|Q|) \) words in total (but may be unevenly distributed). The remaining difficulty is to force the annotator to send the correct timestamps, and to only send one annotation timestamp per element.

We send the timestamps for queries to \( X[i] \) one at a time by stringing together queries, only receiving the timestamp of the next query to an element each time we access it. The annotation protocol for a new element in the stream is:

- Let the new stream element be \( x = X[i] \) or a query \texttt{index}(i). Then the annotator must send the timestamp \( j \) of the next query\footnote{The next query is the future query with the smallest timestamp.} that accesses item \( i \). The client saves element \( x \) for timestamp \( j \) by performing an \texttt{attach}(\( j, i \circ x \)) operation, where \( \circ \) denotes concatenation of words.
– If the stream element is a query \( \text{index}(i) \), the client first performs a \( i' \circ x = \text{recover}() \) query to retrieve element \( x \). We check if \( i' \) and \( i \) match and return \( x \) if so; otherwise we report a protocol inconsistency.

Note that when an item \( x \) is received in the stream, the client can correctly attach it, providing us with the ground truth in the chain of queries to the element. Observe that if the recovered \( i' \) does not match \( i \) for a query, the protocol was broken, as either the annotator told us a wrong future timestep for the next query to \( i \) or the annotated recovery data structure gave a wrong answer. In either case, we have an inconsistency.

3 Streamed Recovery

As previously defined, the \textit{streamed recovery} problem is to maintain a data structure that allow us to attach \((i, x)\) some bitstring \( x \) to the arrival of a stream element at time \( i \), and to recover the bitstring attached to the current stream element. We now give the proof for Theorem 2.

The overall idea in our solution is to maintain a small data structure on the client that is updated when performing attach operations and used to ensure that the answers to recover queries provided by the annotator are correct.

To simplify our presentation, we initially assume that all attach updates are performed first, followed by all recover queries. This assumption can be removed as shown later without increasing the time and space. Remember that \( R \) is the list of items to attach or recover, indexed by the timestamp of the items.

From a high level, we build a balanced B-tree \( T \) with out-degree \( B \) and \( R \) at the leaves, where each internal node have a data structure that allow consistency checking the leaves in its subtree. We let \( T_i \) refer to the leaf corresponding to \( R[i] \). When an \( \text{attach}(i, x) \) operation is performed all nodes on the path from \( T_i \) to the root are updated. By using the consistency checking data structures when a \( \text{recover} \) query is answered, we can check if the answer fit the expectation.

Since \( T \) is balanced with out degree \( B \), it has height \( O(\log_B |R|) \). Each node \( u \in T \) covers some interval of \( R \). We use Karp-Rabin Fingerprints to check the consistency of subtrees, storing for each node \( u \in T \) the \( O(B) \) fingerprints for all prefixes of its children. Clearly, storing \( T \) and the fingerprints takes \( O(|R|) \) space. However, using the annotator we can reduce the space required to \( O(B \log_B |R|) \) as recover queries only move forward in time (so there is no reason to store fingerprints to check the past or the distant future). This is done as follows.

At time \( t \) the client stores the fingerprints on a single root-to-leaf path from the root to \( T_t \) as well as all immediate children of that path. This path is called the active fingerprint path and denoted \( A_t \). The active fingerprint path consist of \( O(\log_B |R|) \) layers of fingerprints with \( O(B) \) fingerprints stored in each layer. Thus, the total space required is \( O(B \log_B |R|) \) at any time \( t \).

Since time moves forward, the active fingerprint path starts at the leftmost root-to-leaf path of the tree and moves right through the leaves, each of which correspond to a single \text{recover} query. The fingerprints in \( T \) are constructed by the client when
performing \texttt{attach}(i,x) operations. The details are as follows. Note that for any \( t \) the active fingerprint path moves from \( T_t \) to \( T_{t+1} \) through a \textit{diamond transition path}: the path from \( T_t \) to \( T_{t+1} \) pass through a common ancestor \( p \) and the two leafs are the rightmost and leftmost leaf below two neighboring children of \( p \), respectively. Since the path moves left to right we can find \( p = \text{lca}(T_t, T_{t+1}) \) on \( A_t \). Let \( v_t \) be the child of \( p \) that \( A_t \) passes through. We know there is a right neighbour \( v_{t+1} \) of \( v_t \) that \( A_{t+1} \) must pass through. At time \( t \) the client stores all fingerprints for children of \( p \) and thus the full fingerprint for \( v_{t+1} \). That is, \( p \) stores the \( O(B) \) fingerprints for its children. Thus, we can force the annotator to send us the correct list of leaves below \( v_{t+1} \), checking the received items with our stored fingerprint. Furthermore, at the same time as receiving these items, we can build up the leftmost fingerprint path in the subtree rooted by \( v_{t+1} \). That is, when we move the active fingerprint path, we can make sure that the annotator sends us the correct list of leaves below \( v_{t+1} \).

The annotation as described transmits each leaf \( O(\log_B |R|) \) times, as it is sent once for each ancestor node in \( T \). However, a lot of annotation may be sent per stream element. We can ensure \( O(\log_B |R|) \) annotated words per element with the following deamortization. The annotator must repeatedly send the items that should be recovered by the right neighbour node on each level of the tree. Transmission of the leaves in the subtree rooted by \( v \) is timed such that it ends when the active fingerprint path moves to \( v \). That is, transmission of the leaves below \( v_{t+1} \) begin when the path moves to node \( v_t \), where \( v_{t+1} \) is on the right of \( v_t \) and in the same level. The result of this is that we at each timestamp transmit \( O(1) \) leaves for each of the \( O(\log_B |R|) \) levels in the tree.

Our final concern is to remove the requirement of non-interleaved attach and recover queries. In this case, we build \( T \) incrementally. This means that we may have already transferred the leaf \( T_i \) (as an empty leaf) before an \texttt{attach}(i,x) operation is performed. It is up to the client to later correct the fingerprint for the ancestors of \( T_i \) when an \texttt{attach}(i,x) is made (so we can check recover queries correctly). This can be done by the client assuming the client can already decide where to \texttt{attach}(i,x) items: it involves updating the fingerprints already stored and in transfer that covers item \( i \), of which there are at most \( O(\log_B |R|) \).

\begin{flushright} \Box \end{flushright}

\section{Compressed Pattern Matching}

We now show how to use the power of the annotator to solve the pattern matching problem in a phrase-compressed text stream. The compressed phrases are on the form \( i = (j, \sigma) \), which either extends a previous phrase \( j \) by an alphabet character \( \sigma \), or starts a new chain of extending phrases if \( j = -1 \). This models most phrase-based compressors, such as LZ78.

The problem is to find occurrences of an (uncompressed) pattern \( P \) of \( m \) alphabet characters in the uncompressed text as follows: for each arriving phrase we must output true iff there is an occurrence of \( P \) ending in the latest phrase. At any time \( n \) denotes the number of phrases we have seen so far.

In compressed pattern matching, the output when phrase \( n \) arrives can depend on phrases arbitrarily far in the past. This is in contrast to well-studied uncompressed
streaming pattern matching problems in which the output typically only depends on a window of length $O(m)$. More formally, in Section 5 we prove that in the standard, unannotated streaming model, there is a space lower bound of $\Omega(n)$ for the pattern matching problem in a phrase-compressed text stream. We also prove that this lower bound holds in the restricted case that stream is LZ78 compressed [26].

We first give an algorithm which determines whether $P$ is a substring of the latest phrase, we will then extend this to find new pattern occurrences that cross phrase boundaries (but still end in the latest phrase). Before we do so, we briefly discuss some (mostly) standard pattern matching data structures that we store on the client for use in for our solution.

**Additional client-side data structures.** The following standard data structures use $O(m)$ space on the client and can be constructed during preprocessing in $O(m)$ time using standard methods.


We also build a set of $m$ perfect static dictionaries. Each dictionary $D_j$ is associated with a pattern prefix $P[0,j-1]$. The role of these dictionaries is to provide an analogue of the classic KMP prefix table. However, unlike the classic prefix table, this approach will lead to worst case constant processing times. Each entry in $D_j$ is keyed by an alphabet symbol $\sigma$ and associated with a length $\ell$. Here $\ell$ is largest non-negative integer such that $P[0,\ell-1] = P[j-\ell+1, j-1]$ and $P[\ell] = \sigma \neq P[j]$. The dictionary $D_j$ contains every symbol for which $\ell$ is well-defined. This construction was also used in [6] (see Lemma 1) which is in turn a rephrasing of the original approach from [23]. It was proven in [23] that, surprisingly, the total size of all $D_j$ summed over all $j \in [m]$ is only $O(m)$. Our perfect static dictionaries are constructed according to the FKS scheme [11] so lookup operations take $O(1)$ worst-case time.

**Occurrences in the latest phrase.** We first give a simple $O(n+m)$ client space and $O(1)$ time solution which does not use annotation. When each phrase $i = (j, \sigma)$ arrives, the output is $\text{match}(i)$ which is $\text{True}$ iff there is a match in phrase $i$. To determine this we also calculate the length of the longest suffix of phrase $i$ that matches a prefix of $P$. This length is denoted $\text{pref}(i)$. We store both $\text{pref}(i)$ and $\text{match}(i)$ for each phrase seen so far in $O(n)$ space.

We compute $\text{pref}(i)$ from $\text{pref}(j)$ and $\sigma$ using the dictionary $D_j$, in a similar way to the KMP algorithm. In particular if $\sigma = P[\text{pref}(j) + 1]$ then $\text{pref}(i) = \text{pref}(j) + 1$. Otherwise, we look up $\sigma$ in $D_j$ in $O(1)$ time. If $\sigma$ is in the dictionary then $\text{pref}(i) = \ell + 1$. Otherwise, $\text{pref}(i) = 0$. This follows because both $P[0, \text{pref}(j)]$ and $P[0, \text{pref}(i)]$ are pattern prefixes.

To decide whether a match occurs in phrase $i$, we make the observation that,

$$\text{match}(i) = \text{True} \text{ if and only if } (\text{pref}(i) = m \text{ or } \text{match}(j) = \text{True}).$$

---

2 We assume a linear alphabet size.
This follows because a match in phrase $i$ is either completely contained in phrase $j$ (in which case $\text{match}(j) = \text{True}$) or ends at the final character of phrase $j$ (in which case $\text{pref}(i) = m$). This completes the basic algorithm description.

To reduce the space to $O(B \log B n + m)$ we use the annotated recovery data structure that we gave in Theorem 2. From the description, we spend $O(n)$ space storing the facts $\text{pref}(i)$ and $\text{match}(i)$ for each phrase, while the client side data structures for processing phrases take $O(m)$ space in total. We reduce the first term to $O(B \log B n)$ by the observation that for a new phrase $i = (j, \sigma)$ we only ever require access to $\text{pref}(j)$ and $\text{match}(j)$, each of which can be obtained from the data structure of Theorem 2 using recover queries. Furthermore, when receiving phrase $i$ the can send the timestamps of all phrases directly extending $i$ in the annotated stream for use in attaching facts to these timestamps. Since a phrase $i$ may be extended multiple times, the trick is to avoid sending the timestamps for all phrases that extend $i$ at once.

We force the annotator to string together the extension timestamps as in the annotation scheme for multi-indexing as follows. The annotator must send two indices for phrase $i = (j, \sigma)$: the index $j'$ of the next phrase extending $j$ and the index $i'$ of the first phrase extending $i$. We attach $\text{pref}(j)$ and $\text{match}(j)$ to phrase $j'$, calculate $\text{pref}(i)$ and recover queries are made in total and all phrase processing besides the operations of Theorem 2 take constant time and $O(m)$ space, we obtain Theorem 4 below.

**Theorem 4.** In a stream of $n$ phrases of compressed text, one can find the phrases that contain a pattern of length $m$ in $O(B \log B n + m)$ space, $O(\log B n)$ worst case words of annotation per phrase and $O(\log B n)$ time per phrase.

We now give our main result by showing how to extend this to find matches that cross phrase boundaries. To simplify exposition, matches within phrase boundaries are still found as shown above. We will give our full algorithm in two versions. First we give a time efficient, $O(n + m)$ space solution which is based on existing techniques for the classic offline, unannotated version of the problem. Then we improve the space to be logarithmic by using our annotated recovery data structure.

**Occurrences across phrase boundaries.** In our first solution when phrase $i$ arrived we computed $\text{pref}(i)$, the longest suffix of phrase $i$ that matches a prefix of $P$. In addition we will also compute $\text{tpref}(i)$, the longest prefix of the pattern matching a suffix of the text up to and including phrase $i$. This is distinct from $\text{pref}(i)$ because it allows for prefix matches that cross phrase boundaries. In particular it is (conceptually) more difficult to compute because the prefix may cross many phrase boundaries. It is important to observe that for any phrase $i = (j, \sigma)$, in contrast to our previous in-phrase approach, a cross-boundary match in phrase $i$ is not generally implied by $\text{tpref}(j) = m$. This is because $\text{tpref}(j)$ can extend to the left of phrase $j$ and into a region of the text unrelated to phrase $i$.

To enable us to find cross-boundary matches efficiently, we will also compute $\text{sub}(i)$, the length of the longest substring of the pattern which matches a prefix of phrase $i$. 
We also store the location of an occurrence of this substring in $P$. In the first version we explicitly store $\text{pref}(j)$, $\text{tpref}(j)$ and $\text{sub}(j)$ (and its location) on the client for all phrases seen so far in $O(n)$ space.

The key observation is that when $i = (j, \sigma)$ arrives any new cross-boundary matches are completely contained within the portion of the text given by concatenating the strings corresponding to $\text{tpref}(i - 1)$ and $\text{sub}(i)$. This follows immediately from the definitions of $\text{tpref}(i - 1)$ and $\text{sub}(i)$. These are both substrings of $P$. Further, we can compute $\text{sub}(i)$ from $\text{sub}(j)$ (and its location) and $\sigma$. This can be done in $O(1)$ time straightforwardly using the suffix tree for the pattern as follows. If $\text{sub}(j)$ is shorter than the length of phrase $j$ then $\text{sub}(i) = \text{sub}(j)$.

Otherwise we must decide whether phrase $i$ is a substring of the pattern. This in turn can be achieved as follows. Recall that for any $\text{sub}(j)$ we store the location of a corresponding substring in $P$. In-fact we store this location as a reference to the node in the suffix tree for $P$ which represents this substring. We can then determine whether $\text{sub}(i) = \text{sub}(j)$ or $\text{sub}(i) = \text{sub}(j) + 1$ in constant time by looking for $\sigma$ on an edge leaving that node. Note that determining $\text{sub}(i)$ does not identify any new matches besides $\text{pref}(i)$ by itself, as if $|\text{sub}(i)| = m$ then we must have $\text{pref}(i) = m$ as well.

As observed above, any new cross-boundary matches are contained within the concatenation of two pattern substrings. Therefore, we can apply Lemma 1 to find the substring cross-boundary matches in $O(1)$ time.

Lemma 1 (Gawrychowski [12], Lemma 3.1 rephrased). The pattern $P$ can be preprocessed in $O(m)$ time and $O(m)$ client space to allow queries of the following form in $O(1)$ time: Given two substrings $P[i_1, j_1]$ and $P[i_2, j_2]$ decide whether $P$ occurs within $P[i_1, j_1] \odot P[i_2, j_2]$.

We now explain how to use our annotated recovery data structure to again improve the space used on the client to $O(B \log_B n + m)$. As previously shown, we can attach and recover the required facts $\text{pref}$ and $\text{match}$ for each new phrase $i = (j, \sigma)$ by stringing together the facts using annotated knowledge about the next references to $i$ and $j$. We store the new facts $\text{tpref}$ and $\text{sub}$ in the exact same way, attaching the them to the next references to $i$ and $j$. As we spend $O(n)$ space only on the facts and $O(m)$ on the remaining data structure, this concludes the full algorithm description and our proof of Theorem 1.

5 Lower bounds.

In this section, we prove that in the standard, unannotated streaming model, there is a space lower bound of $\Omega(n)$ for the pattern matching problem in a phrase-compressed text stream. This follows via a reduction to the indexing problem with the pattern $P = 1$.

We can use the first $n$ phrases to encode a bit string of length $n$ (by starting a new chain with every phrase). We can then ‘query’ any of these bits by appending the phrase $(j, 0)$ where $j$ is the index to be queried.

3 No new symbol make a longer substring of $P$ match as there is a previous mismatch.
We now consider the restriction to a LZ78 compressed text stream, instead of a more general phrase-compressed text stream. Under LZ78 compression, the phrases given for the lower bound above are not a valid compression of the underlying string. In particular in LZ78 compression there cannot be two phrases which correspond to the same substring (except if one of them is the final phrase). This is because LZ78 phrases are defined to be maximal. That is when compressing a text the next phrase will represent the longest (valid) prefix of the uncompressed suffix of the text.

With an unbounded alphabet, a suitable lower bound for LZ78 is given as follows. The overall technique of reduction to indexing is the same. Start with $n$ phrases each representing a single different symbol $x_1 \ldots x_n$. Then encode the bit string $B$ one bit at a time, with the $i$-th bit encoded as the phrase $2i = (i, B[i])$. The pattern is $P = 1$. A query to bit $j$ is modeled by a phrase given by $(3j, 0)$. If there is a match we know the bit was a 1, otherwise 0. Similar (but slightly more involved) constructions work for binary alphabets.

References