Natural Language Engineering **19** (1): 61–93. © Cambridge University Press 2012
doi:10.1017/S1351324911000349

A fast and flexible architecture for very large word n-gram datasets

MICHAEL FLOR
NLP and Speech Group, Educational Testing Service, Princeton, NJ 08541, USA
e-mail: mflor@ets.org

(*Received 11 April 2011; revised 23 November 2011; accepted 30 November 2011; first published online 10 January 2012*)

Abstract

This paper presents *TrendStream*, a versatile architecture for very large word n-gram datasets. Designed for speed, flexibility, and portability, TrendStream uses a novel trie-based architecture, features lossless compression, and provides optimization for both speed and memory use. In addition to literal queries, it also supports fast pattern matching searches (with wildcards or regular expressions), on the same data structure, without any additional indexing. Language models are updateable directly in the compiled binary format, allowing rapid encoding of existing tabulated collections, incremental generation of n-gram models from streaming text, and merging of encoded compiled files. This architecture offers flexible choices for loading and memory utilization: fast memory-mapping of a multi-gigabyte model, or on-demand partial data loading with very modest memory requirements. The implemented system runs successfully on several different platforms, under different operating systems, even when the n-gram model file is much larger than available memory. Experimental evaluation results are presented with the Google Web1T collection and the Gigaword corpus.

1 Introduction

N-gram language models (LMs) have become increasingly popular as a common resource for data-driven algorithms in many areas of computational linguistics. Along with traditional use for speech recognition, n-gram models have been embraced for a variety of text-processing tasks, such as statistical machine translation, text tagging, spelling correction, named entity recognition, word sense disambiguation, lexical substitution, etc. (Lapata and Keller 2005; Bergsma, Lin and Goebel 2009).

Availability of larger and larger text corpora and use of the web as a corpus allow for development of very large word n-gram models based on many millions of counts and more (e.g., *Google Web1T*; Brants and Franz 2006). Furthermore, the typical maximal n-gram length in such collections has grown from three to five, and models with higher order n-grams are emerging (Sekine 2008). This leads to n-gram databases with billions of n-grams and multi-gigabyte storage sizes (Sekine and Dalwani 2010).

Wider adoption of massive n-gram models, both in research and in industrial settings, may require improved performance, simplicity, and flexibility on the part of

software systems that support n-gram storage and retrieval. This leads to several sets of considerations. An obvious consideration is search speed: given a searchable store of n-grams, how fast is the search? For exploratory linguistic research, obtaining n-gram query results within several seconds or even minutes can be satisfactory. For Natural Language Processing (NLP), some applications can be happy with a network- or web-based n-gram query service. However, in some applications, very fast local n-gram searching is crucial (for example, when network service is too slow or unavailable, or when huge numbers of n-grams queries need to be served extremely quickly). But speed is not the only consideration. Additional considerations stem from aspects of portability, ease of deployment, and utilization conditions. Portability becomes simpler if the same n-gram database can be used on a variety of platforms. Deployment considerations become less complicated if the same n-gram database can flexibly operate with different CPU and memory resources. Running an n-gram model on a dedicated server has its advantages, but adoption of n-gram models may become more widespread if such database could also run on modest hardware, concurrently with other applications.

Another set of aspects relates to ease of generating a compiled (binary-format) searchable store for a client application. The basic aspect is: how much time, RAM and CPU power does it take to compile a pre-tabulated very large collection of n-grams into some sort of fast-searchable format. Moreover, it can be quite useful if the same toolkit can also generate and compile an n-gram database directly from text corpora. Such capability may allow fast generation of n-gram models for different content domains and for specific purposes. Another aspect of generation is versioning: How easy is it to generate a new compiled searchable version of the data – for example, with different filters, for example, with or without punctuation. A third aspect is updateability – can we update the compiled searchable store with additional data, or do we have to regenerate everything from original (textual or tabulated) data? The need for adaptive (updateable) language models arises in certain contexts, for example, when data are accumulated over time (Levenberg and Osborne 2009; Wang and Li 2009), or when combining data from multiple sources.

This paper presents TrendStream, an n-gram storage and retrieval system engineered for flexibility and speed. Our design was informed by a set of practical requirements: ease of deployment, fast system activation, fast search for literal n-grams, modest, and controllable memory utilization, flexible enough to use lots of memory if available (a system should run on high- and low-end platforms, even laptops), lossless compression, and support for updating the stored model. The resulting architecture also supports pattern-matching queries without any additional indexes.

The rest of the paper is structured as follows. Section 2 provides a brief overview of prominent architectures for large n-gram models, Section 3 gives a theoretical and technical description of the TrendStream system, Section 4 presents some empirical results for a complete implemented system. Section 5 presents a brief comparison with other systems and also discusses some additional features of the system.

2 Architectures for n-gram models

2.1 A variety of approaches

A variety of techniques for storing and deploying very large n-gram models have been recently described in the language modeling literature. Approaches vary with the underlying requirements for speed and memory use, hardware availability, and field of application. Brants *et al.* (2007) and Emami, Papineni and Sorensen (2007) describe methods for processing very large language models on distributed server clusters. We restrict our attention to architectures that can be deployed on a single machine, since this is still the more practical and simpler approach in many cases. For deployment on a single machine, n-gram database architectures fall into three main classes: (a) file-based indexing, (b) relational databases, and (c) specialized systems (LM toolkits).

Following the release of the Google Web1T dataset (Brants and Franz 2006), some deployment solutions for this huge collection (88GB, or 25-GB gzip compressed) utilize the original flat files with additional inverted indexing for binary search (Yuret 2008; Giuliano 2007). Islam and Inkpen (2009a) describe a file-based inverted index approach for managing all 5-grams of the Web1T collection. A sophisticated method to speed up file-based index lookup via hashing is described by Hawker, Gardiner and Bennetts (2007). Ceylan and Mihalcea (2011) have presented a very efficient indexer for large n-gram corpora. This system uses a B+ tree for compactly holding an index to the set of flat files and shows best retrieval speed among the file-based approaches. As a search/retrieval method, file-based indexing over original tabulated n-gram data tends to be relatively slow (compared with LM toolkits). Depending on implementation, such approaches are also restricted in the kind of queries that can be easily processed – typically supporting only literal queries. Systems by Hawker *et al.* (2007) and Ceylan and Mihalcea (2011) do allow wildcard capabilities at the expense of slower performance or a huge additional index. With file-based indexing, limitations of portability may arise when deploying such systems on a different platform. Finally, this approach is not well suited for data updating, for example, incrementally generating very large n-gram collections from text documents ('streaming' data). The indexes need to be regenerated when data are modified, and, since data are constantly updated, in practice, indexing is done only when data are finalized.

Storing a language model in a relational database is a viable solution for large datasets. Carlson and Fette (2007) report on using the Web1T stored in a MySQL database on a dedicated server. Advantages of this approach include rich SQL-based querying capabilities, support for wildcard queries (with additional indexing); and inherent support for updating with more data. Disadvantages of this approach are a need for a powerful platform (possibly requiring a dedicated server), restricted portability between different platforms, and potential license, and maintenance costs for some Relational Database Management System (RDBMS). Evert (2010) describes a very promising architecture – storing and indexing the Web1T collection with a set of Perl scripts and an embedded SQLite database suitable for deployment on servers and some desktops. Populating a modern RDBMS with more than a billion of n-grams and generating applicable indexes may take quite a long

time – Evert (2010) reports two weeks indexing on a 16-GB RAM server. Query performance speed with RDBMS can be adequate for exploratory linguistic research, but may fall short of the needs of intensive NLP applications. For example, Guthrie and Hepple (2010b) report that MySQL 5.0 on a 64-GB RAM server achieves the rate of processing 527 queries/second (on a dataset of about 56 million bigrams and unigrams), while modern LM toolkits process many thousands of queries per second.

Originating from earlier use of n-gram models for speech recognition, custom-engineered encoding approaches (LM toolkits) focus on speed of query processing and minimizing the size of language models in memory (Stolcke 2002). To facilitate fast processing, n-gram models 'should ideally reside locally in memory to avoid time-consuming remote or disk-based look-ups' (Talbot and Osborne 2007a). Such requirement poses a challenge for language models that serve very large n-gram collections. Accomplishing this is the current trend of LM architectures. Yet, loading a very large model into memory is sometimes unsuitable, even on platforms that do have enough memory – for example, if there is a need to run other applications concurrently on the same platform.

Two principles have guided the development of modern optimized LM architectures: (1) providing efficient data structures for encoding/retrieval, (2) use of compression methods to minimize the storage and memory requirements. Given that language models are basically mappings from strings (keys) to 'payload' data, optimizing a large language model involves two orthogonal directions: (a) compress the set of n-grams stored in the model, and (b) compress the payload – the frequency counts (or probability values) and the back-off weights stored for each n-gram entry. Compression and optimization are typically tightly related to the core data structure for a given architecture, but there are some common principles.

One widely used approach for compressing the payload is quantization – a lossy compression method that maps values from a continuous range onto a discrete set. During encoding of an n-gram model, the actual count and back-off values associated with each n-gram are replaced with quantized values from a limited set, thus reducing the storage requirements. A simple and common approach is 8-bit quantization, which allows for 256 distinct values. Different methods of dividing the continuous range of values into quantized sets were explored (Whittaker and Raj 2001; Federico and Bertoldi 2006). Notably, quantization is a light obstacle for updateability – when data are added or updated, a quantization scheme may be compromised and the whole dataset may need to be re-quantized. Still, quantization remains one of the most widely used methods for reducing the storage requirements (Harb *et al.* 2009).

A different perspective is to ask what payload data is actually useful. Recently, Brants *et al.* (2007) have shown that smoothing (and therefore storage of back-off values) is not useful for n-gram collections based on more than a billion-word tokens. Thus, some recent LM toolkits support both variants (Pauls and Klein 2011).

Issues of storage and speed optimization play major role in describing the two most prominent modern architectures for very large language models – prefix trees and randomized hashing.

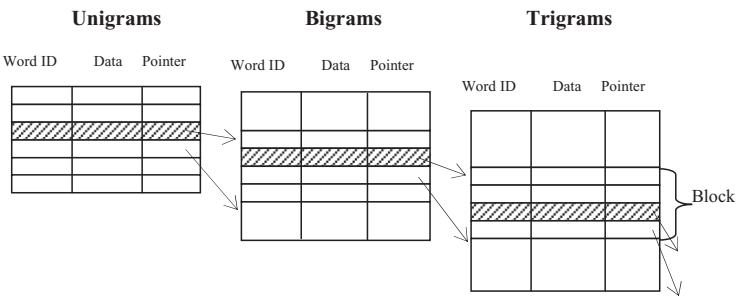

Fig. 1. Generic data structure for a trie, with word-level granularity and words on level i+1 grouped into blocks by shared prefix on level i.

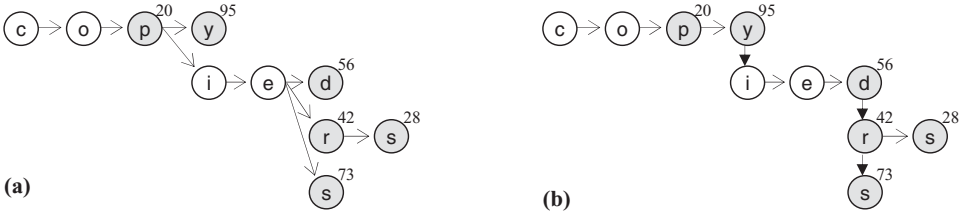

Fig. 2. (a) Schematic character-level trie for a set of words (with fictional counts). (b) Same trie as a linked-list data structure (with: syntagmatic → and 'alternative' ↓ links). End-of-word nodes are shaded for illustration.

2.2 Prefix trees and n-gram models

A prefix tree (digital trie; Fredkin 1960) is a common data structure for compact storage and efficient search of symbol sequences that have common prefixes. Each symbol in sequence is represented by a node in a tree structure. Nodes are connected via directed arcs and the sequence of arcs from common root uniquely defines each sequence (see Figures 1 and 2). When associated data are stored on trie nodes, trie serves as an indexing structure, with token sequences as search keys.

For language modeling, a classic design represents a model as a prefix tree with word-level nodes (Whittaker and Raj 2001; Raj and Whittaker 2003). The tree structure originates from a (virtual) root node, which branches out into unigram nodes, each of which branches out to bigram nodes at the second level and so on. Each word-node in the tree represents an n-gram that ends with that word, with a context represented by the sequence of words from the root of the tree. Each node has a fixed record structure that stores the word ID, associated value(s), and references to 'next order' word-nodes (successors). To improve memory locality during traversals and to reduce memory footprint, for any given node on depth *i*, child nodes at depth *i+1* are typically grouped together into a block and stored in an array sorted by word ID.

For vocabularies with less than 65,536 words, 16-bit values are sufficient for numeric word IDs; larger vocabularies require 32-bit integer values. Various methods have been proposed for compact representation and compression of word IDs and associated data (Harb *et al.* 2009). Raj and Whittaker (2003) describe a compression

method where word IDs are stored in sorted arrays, and data values (probabilities) are compressed via quantization. For very large n-gram models, Federico and Cettolo (2007) use variable length records for each node. Germann, Joanis and Larkin (2009) use variable length encoding for both word IDs and probability values. Watanabe, Tsukada and Isozaki (2009) use word-level trie architecture and compress it in a succinct data structure that represents a static trie as a continuous bit string.

Pauls and Klein (2011) present a system with several variants: one based on sorted arrays, another one extends this with variable length encoding of the data, and a third one uses hash tables instead of sorted arrays. Heafield (2011) presents a system with two variants. One is a reverse trie (storing n-grams in reverse word order) with sorted arrays, and the other uses an advanced implementation of hashing techniques. Both systems exhibit outstanding compression and very fast n-gram searching.

Instead of loading a whole model into memory, portions of a model can be stored on disk and loaded into RAM as they are required. For trie-based architecture, a natural unit for on-demand loading is the block of successor nodes (see Figure 1; Ravishankar 1996). If a block of successors is stored as a contiguous region in file, it can be indicated by relative offset value(s). A relevant file-section can be loaded to RAM via memory-mapping (Federico and Cettolo 2007; Zens and Ney 2007). Then, looking for a specific successor word requires a binary search in a sorted array.

Another architectural issue is the use of mutable versus serialized representations. Mutable data structures represent tries in memory as collections of interlinked node objects (or records), using object references or memory pointers. Such implementation offers fast lookup and flexibility. For storage on disk, such data structure has to be serialized. A typical serialization is a contiguous array of records where reference pointers are substituted with offset values. Germann *et al.* (2009) propose a compression scheme where the whole model is encoded as a serialized array with variable length encoding. For search, the file is memory-mapped and traversed in RAM, without unfolding it into a mutable trie. Locally compressed data and offset values are decompressed on the fly in course of a search. However, such n-gram models are not updateable and must be loaded whole in memory (posing considerable memory requirements).

2.3 Randomized hashing for storing n-gram models

Randomized language models use random hash functions to map n-gram strings to their associated counts/probabilities. The basic idea is that the size of the stored model can be significantly reduced if the n-gram strings are not stored at all. Instead, hash values of n-gram strings are used as keys. During encoding, a set of random hash functions is used to represent each n-gram as a bit vector and all such vectors are stored together in a specialized data structure. During retrieval, the same hash functions are applied to a query n-gram and a bit vector is sought in the data structure. A huge reduction in storage size is achieved by the specialized data structure that stores all the bit-vectors in a massive, randomly overlapping way (many bits are shared by different keys). Talbot and Osborne (2007a, 2007b) implemented this approach with the Bloom filter (Bloom 1970) as a basis for the specialized

data structure. Talbot and Brants (2008) describe a system using the Bloomier filter (Chazelle *et al.* 2004). Guthrie and Hepple (2010a, 2010b) proposed an even more compact representation that uses minimal perfect hashing. For a concise technical description of randomized hashing, see Guthrie and Hepple (2010a).

One drawback of randomized hashing architectures is the possibility of errors. Methods based on the Bloom filter may sometimes return incorrect count for stored n-grams (Guthrie and Hepple 2010a). All randomized hashing architectures have false-positive errors: For an n-gram that was never stored in the model, a system may return some value instead of reporting that the n-gram was not found. There are methods to minimize such errors by storing some small additional information for each n-gram (the so-called 'fingerprints'). By controlling the size of "fingerprints", probability of errors can be adjusted, but this increases the size of the stored model.

Since randomized hashing architectures do not store the actual n-grams, the n-grams in a model cannot be enumerated. While this is not a requirement for applications that only need handling of literal queries, it does mean that wildcard queries cannot be supported with such architecture, at least in its current form[1].

For operational deployment, purely hash-based language models must be completely loaded in memory. This carries an advantage and a disadvantage. All queries are served in RAM, so performance is quite fast. The disadvantage is that such architectures do not allow partial loading. Therefore, even if hash-based architecture can store a language model in significantly less space than trie-based approaches, it still imposes considerable limitations for hardware. For example, one of the methods described by Guthrie and Hepple (2010b) may store the entire Web1T collection in about 10 GB, but it also means that a computer will need at least that amount of RAM available to work with such model[2].

Another aspect of randomized hashing architectures concerns updateability – adding new n-grams and updating counts for stored n-grams. Talbot (2009) and Van Durme and Lall (2009) have demonstrated methods for approximate counting (count updates) in randomized hashing models. In addition to probabilistic storage of n-grams and counts, these solutions introduce data structures to support probabilistic updates, which are lossy in the sense that they may lead to estimation errors (reporting inaccurate counts). Here again, probability of errors can be lowered by assigning more memory to the probabilistic counters, which increases the memory footprint of the whole encoded model.

Architecture presented by Guthrie and Hepple (2010a, 2010b) does allow changing of counts, so it is partially updateable. Yet, like the other hashing-based methods, it

[1] One can imagine precompiling the counts for at least some of the wildcard queries as distinct entries. For example, 'the_*_barked' can be stored as a 'literal' entry associated with the combined counts of all trigrams that start with 'the' and end with 'barked'. This can be useful, but such approach is quite limited and cannot provide comprehensive wildcard search capabilities.

[2] With additional engineering, the necessity to load the whole model into memory can be circumvented. For a very large model, one could construct a set of hashes, for example, a separate randomized hash for n-grams starting with a given letter, or for n-grams of given length, and then dynamically load parts of the whole model as separate hash structures. A compartmentalized approach may also bring additional benefits such as less collisions in the hash and shorter fingerprints for error correction.

does not allow insertion of new keys after compilation. Variants of Minimal Perfect Hash Rank (MPHR) use quantization based on corpus statistics – this allows very tight compression, but precludes updateability of counts – because updating would change corpus statistics and would require re-encoding of the whole dataset.

Static randomized hashing approaches cannot incorporate new data without fully retraining from scratch (Levenberg and Osborne 2009). An inherent property of randomized hashing models is that the size of the full set of keys (number of distinct n-grams) must be available (or estimated) when the randomized hashing data structure is constructed – to achieve adequate hashing and prevent collisions. This sets a limitation on growth of the model – the model can be populated with new n-grams only up to a pre-estimated limit, beyond that new n-grams cannot be added to the model (adding new keys will compromise the hashing). Efficient estimation of hashing parameters *a priori* (without scanning the target dataset) remains an open research question (Talbot 2009). The choice of parameters also predefines model size – the model is constructed *a priori* to accommodate a certain number of distinct n-grams and the corresponding memory footprint is reserved, rather than growing incrementally with population. Levenberg and Osborne (2009) present a dynamic variant of randomized hashing, one that allows inserting new material into a language model (and even deleting some stored data). This is achieved by introducing an overflow data structure that stores data beyond the predefined capacity of the core randomized structure, and relying on the assumption that if the core data are trained on a large corpus, truly new n-grams will be rare and can thus be accommodated in the overflow.

Static randomized hashing models do achieve excellent compression. Some comparisons of such models with trie-based models are presented by Heafield (2011) and Pauls and Klein (2011).

3 TrendStream system design

This section provides a detailed description of the TrendStream architecture. Notably, TrendStream was designed with the goals of fast activation, memory flexibility, ease of deployment, and incrementally updateable data structure. The TrendStream architecture is based on the forward trie data structure. We introduce several novel features in this design: (a) character-level and byte-level granularity of prefixes, (b) utilization of ternary prefix trees (Bentley and Sedgewick 1997) instead of sorted arrays, (c) a new method for individually compressing each n-gram record, and (d) a hybrid approach to combine mutable in-memory data structures with serialized compressed records.

3.1 The choice of granularity

Word-level granularity is traditionally preferred for trie-based word n-gram models because it is considered more memory-efficient, especially when words are represented with numeric IDs. However, more sharing of common prefixes is possible with character symbols. Instead of using word-level symbols (Figure 1), we opted for

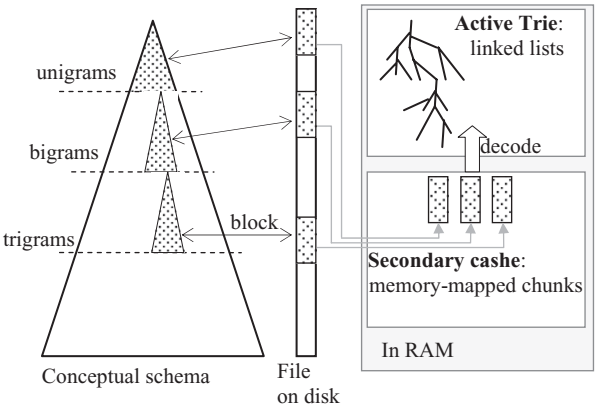

Fig. 3. Schematic outline for word n-gram trie, with character-level granularity. Shaded triangles represent sub-trees.

character-level granularity (Figure 2). With character-level encoding for n-grams, we must explicitly encode inter-word delimiters (we use the '_' symbol). Yet, with adequate compression, an efficient and flexible character-level trie for large word n-gram models is feasible.

3.2 System structure

The TrendStream architecture is outlined in Figure 3. Conceptually, the whole trie for a large n-gram model is a tree of linked nodes with a single root. We use a simple forward trie: n-grams are stored without any rotations. Each node represents one character (but see Section 3.8, which describes a node-per-byte correspondence).

Only a small portion of the trie is active in memory at any time ('Active Trie'). The whole trie is stored on a disk in a serialized compressed form. The whole file, or portions of it, can be loaded into memory (via memory-mapping), and only relevant parts of the memory-mapped chunks are unfolded (decompressed) into mutable linked-nodes structure, a process that is automatically guided by the trie search algorithm (see next section).

The use of uncompressed mutable data structure in the Active Trie has several advantages. Many searches follow the same prefix paths. Keeping those paths in memory (as kind of cache) allows faster search than uncompressing them for each query. On the other hand, with mutable linked lists, parts of the Active Trie can be dropped – thus allowing programmatic regulation of memory loads. A third advantage is that data in the Active Trie can be easily updated (e.g., adding new n-grams). Thus, when needed, the same encoded n-gram model can be used in read/write mode.

3.3 Node structure and search

For a character-level trie, a simple linked-list implementation for each node may have two links: a syntagmatic link (next character in sequence) and 'alternative'

Table 1. *Fields for a mutable hybrid TST node*

Field	Value
node char ID	1 byte
next child	Null or pointer to node, or 64-bit integer (next-child offset).
hi-child	Null or pointer to node, or 64-bit integer (hi-child offset).
lo-child	Null or pointer to node, or 64-bit integer (lo-child offset).
associated data value(s)	Null or 64-bit integer (count) or a structure for complex data.
self-offset in serialized file	64-bit integer.

link (next character in list, see Figure 2(b)). To facilitate faster searches, we use a Ternary Search Trie (TST) data structure (Bentley and Sedgewick 1997). In TST, each node may have up to *three* children, rather than two, and the inter-node linking structure is analogous to alphabetically sorted order. A TST search compares (by numeric value) the current character in the search string with the character at the node. If the search character is less, the search branches into a child that has a lesser char-value ('lo-child'); if the search character is greater, the search branches to the child that has a higher char-value ('hi-child'). When the search character is equal (a match), the search goes to the 'next child' (syntagmatic link) and proceeds to the next character in the search string. Bentley and Sedgewick have shown that ternary recursive decomposition, applied character-by-character on strings, is a very efficient search strategy, with search time linear in the length of the search key.

The classic TST is well suited for in-memory implementation (Bentley and Sedgewick 1998). Here we describe a method to efficiently load portions of the trie from serialized form. We use unidirectional linking, so any child-node is logically accessible only from its parent node. In our Active Trie, a node has a hybrid structure (see Table 1) – it has in-memory references to its children, and also carries the values of children's offsets in the serialized data structure (e.g., on disk). This hybrid structure is the key for flexibility.

When a node is loaded into the Active Trie, it carries the offset values of its children. Any child-node, when requested for the first time, is loaded by the offset value. Upon loading a child, the parent node drops the child's offset value and stores a pointer to child-node. When a child is unloaded, the parent will store the child-offset value again – this allows reloading a child when necessary. The offset of the root node is kept in the header of the trie file, and the root node is always the first one that is loaded. Any other nodes are loaded into the Active Trie only when required by the search process.

3.4 Counts versus probability values

The core data associated with every n-gram in a model are the count (whole integers) or probability (floating-point numbers). The advantage of counts is that they are easily and locally updatable. Our implementation stores a count value for every n-gram. Cumulative counts are maintained for each n-gram order (e.g., 1–5, or higher) and are stored in the file header. Fractional probability values, when required, are

Table 2. *Serialized record structure for a TST node*

Content	Size	Status
1. Indicator flag	1 byte	Mandatory
2. Character value	1 byte	Mandatory
3. Next-child offset	1–8 bytes	Optional
4. hi-child offset	1–8 bytes	Optional
5. lo-child offset	1–8 bytes	Optional
6. Data flag	1 byte	Optional
7. Data value	1–8 bytes +	Optional

Note: data sizes are 'conceptual', actual nodes are tightly compressed with variable length encoding.

computed on the fly during query processing. This design allows easy encoding of the existing tabulated n-gram collections and also generation/update of n-gram models from text.

3.5 Serialized data structure and compression

The serialized structure of a single node derives from the TST node structure described above. Essentially, we serialize a linked-list structure where each node may have up to three links to children. Links are represented by offset values. For each node, we store character value, offset values of its children and the associated data (count value, though more complex data is also used sometimes, see Table 2 and also Section 3.8). All nodes are encoded, in post-order, into a continuous sequence of records on disk. For very large n-gram models, the resulting sequence may take several gigabytes (even with compression), hence we use 64-bit offset values.

Compression is used to minimize the overall file size, and to keep the memory-mapped portions small. For pinpoint activation, each record is compressed *separately*, independently of other records. The record length is variable. Only two fields are mandatory – an indicator flag and character ID (1-byte each). Child offset values are encoded only for children that actually exist (no space is reserved for potential children). Coding takes two steps: First, we use simple delta compression – encode only the differences between node's own offset and the children's offsets. Next, we use variable length encoding of integers. However, instead of varying on the bit-level, or using representation in base-128,[3] we use the core byte-level numeric types as afforded by the programming language (i.e., 1-/2-/4- or 8-byte integers) and specify the field lengths in the indicator flag. The single-byte indicator flag indicates several things at once: how many children there are, their roles (hi/lo/next), how many bytes are used to encode each value, and whether the node carries any data or not. This overloading is achieved by using 250 pre-arranged fixed values, while six more values are reserved for special cases. If data is present, we use an additional 1-byte

[3] as is done, for example, in Apache Lucene (http://lucene.apache.org/java/3_0_3/fileformats.html)

indicator flag (which signifies the data type and how many bytes are used), followed by a variable-length encoded count value. When necessary, more complex associated data can be stored in a similar manner.

3.6 Updateability and optimization

When working in updateable mode, counts for stored n-grams may change and new n-grams may be added, thus introducing new nodes and new children for existing nodes in the trie. In TrendStream, all changes always occur in the mutable Active Trie in memory. Serialization works in two steps: (a) basic update, and (b) optional later optimization.

Basic update works as follows: Both new nodes and affected –old nodes are compressed and serialized individually, as described above. Records for new nodes are appended at the end of the file. Updated old nodes can rarely fit into their old tight places in the file. We use two approaches for growing the serialized file. One approach uses a simple 'hop-trace' solution: store an old node in a new location at the end of file and overwrite the record in old location, leaving a hop-trace. A hop-trace includes two components: a flag value indicating that the data is a 'hop' offset, and an offset value to the new location. When n-gram collection grows very big, 'hop' offsets may need 64-bit integer values. Since an 'old' location is often just 3–4 bytes long, in many cases it cannot store a hop-trace. To ensure that any record can be overwritten with a 'hop-trace', a record must have minimal size. Thus, we use the same serialization format in two modes: padded and unpadded. The padded format differs in that any record that is too small after compression is padded with empty bytes up to a minimal required size. A serialized file with hop-traces is fully operational for literal and wildcard n-gram searches, and for further updating. An alternative approach goes without any hop-traces. When an updated node is serialized, it is always written in a new location, and its parent is required to update itself as well. In serialization, child nodes are always encoded before their parents and this prevents catastrophic chains of updates – by the time a parent node needs to be serialized, all of its children and descendants are already accommodated.

For optimization, the system includes a compaction process that recodes a whole file (trie) and eliminates hop-traces and 'dead' sections. The compaction process can also add or remove record padding. Conversion between padded and unpadded formats is simple, always available, and is very fast. Notably, a file is updateable even after optimization.

With this design, incremental encoding of the existing tabulated collections and collecting from streaming text is essentially the same process, with just different methods for reading source data. Examples of the process are given in Section 4.1.

3.7 Loading by offsets and memory-mapping

In TrendStream, decompression of a node does not depend on other nodes. Any node can be individually found in the serialized sequence by offset value, and quickly decompressed. Depending on memory utilization policy, the whole serialized

sequence, or parts of it, can be memory-mapped and kept in RAM (as Compressed Cache) for fast decompression. This section describes how partial on-demand loading is facilitated.

As indicated by the previous research (Ravishankar 1996; Federico and Cettolo 2007; Zens and Ney 2007), a block of immediate successor words for a given word is a natural and efficient unit for on-demand loading. With a character-level trie, a block of successor words is actually a sub-tree (see Figure 3). Sub-trees corresponding to successor-blocks are automatically identified during compaction–optimization stage of encoding. For each block, nodes are serialized and compressed individually, in post-order, with an additional provision: They are grouped together and saved as a contiguous sequence in the trie file. For on-demand loading, the start- and end-of-block offset values are stored in the record of the inter-word delimiter character (the root of the sub-tree).

In on-demand loading mode, efficient loading of blocks is done in the following manner. When the search algorithm encounters an inter-word delimiter in the Active Trie, a file-chunk (the serialized sub-tree) is memory-mapped and is then ready for fast pinpoint decompression of individual nodes. There is no need to decompress all of the nodes in a block – only those are decompressed that are specified by the search process. The Compressed Cache manages all memory-mapped chunks. To optimize memory usage, it tracks the 'usefulness' of on-demand-loaded chunks and unloads chunks that are not used for a while. Note that dropping a sub-tree from the Active Trie is not the same as dropping a memory-mapped chunk from the Compressed Cache. The two complementary components have separate memory policies that contribute to overall system flexibility.

3.8 Word recoding and byte-level granularity

A character-level trie can be made significantly smaller by recoding all words into shorter code words. During model encoding, we sort all unigrams and assign them with integer IDs in decreasing order of frequency (i.e., static pseudo-Huffman coding[4]). Numeric codes are then converted into base-254-byte sequences. The 254 most frequent words are coded as 1-byte code words, the next 64,516 most frequent words are assigned 2-byte code words. More than 16 million additional words can be encoded as 3-byte sequences, which is enough to cover a reasonable natural language vocabulary. For example, an n-gram such as '*with_admirable_style*' may be converted into a much shorter sequence like 'o_*z8H_b7*'. We use base-254 rather than 256 since we reserve 1-byte value for inter-word delimiter '_', and one additional value is reserved for special uses.

The TrendStream algorithms for search, serialization, and compression work with nodes that have byte-value identities. The TrendStream trie structure has byte-level

[4] In adaptive situations, where frequency distribution of unigrams is unknown, we can borrow a known distribution from another model, or assign codes on the first-come basis. This may lead to sub-optimal coding.

granularity. This allows uniform treatment of words or code words – both kinds are treated as sequences of byte-valued nodes.

In practice, we do not recode the first word of any n-gram. Thus, the layer of unigrams in a trie also serves as a dictionary for translating words into code words. Unigrams are encoded as sequences of nodes where each node carries one-character byte value. A node that encodes the last byte of a unigram carries a complex associated data structure, which includes the unigram frequency (count) and the associated code word byte sequence. When TrendStream is queried about any given literal n-gram, it searches each word separately as a unigram, and attempts to retrieve code words for each word of the query (except first). If this step is unsuccessful, the n-gram is not in the model and the search quickly terminates. Otherwise, a code-worded n-gram equivalent (byte-sequence) is assembled and searched for in the trie. Although a recoding step takes time for each query, it allows fast rejections and shorter searches in the trie.

3.9 Supporting pattern matching queries

Pattern matching searches are supported on the core trie structure without any additional indexes. With literal n-gram queries, the related questions are, 'is this word-sequence in the database?' and 'what is its count/frequency?'. With wildcard queries, we usually need to retrieve n-gram strings as well. Iteration over index keys is a principled integral feature for prefix-trees[5]. Since TrendStream stores codeword byte sequences, it needs to be able to convert them back to word strings. The layer of unigrams serves as a dictionary from words to code words, and there is an inverse dictionary from code words to words. This dictionary is stored as a *separate* trie in the same serialized format, in the same file as the core trie, and uses the same load and search methods. This inverse dictionary has the same number of entries as the unigram layer in the core trie. Inverse dictionary is automatically updated whenever a new unigram is inserted into the system.

Technically, the same implementation supports two different syntax modes of wildcard queries. The 'extended' mode allows full regular expression syntax for any n-gram position, for example, '*is_[ae].+|b..*' retrieves all bigrams where first word is 'is' and second word starts with 'a' or 'e', or starts with 'b' and has exactly three characters. The 'simple' mode supports just the two familiar wildcards: '*?*' and '*'. The question mark stands for any character. When used alone, the asterisk stands for any single word. When used in a pattern with letters and '*?*', the asterisk stands for any sequence of zero or more characters. For example, a query '*a_g?*_*' would retrieve all and only trigrams where the first word is '*a*', the second word starts with two letters, of which the first must be '*g*', the third word may be any word (query matches, for example, '*a_good_day*').

Interpreting a wildcard query involves counting the number of components (separated by inter-word delimiters, for example, blank space or '_'). Each component

[5] Unless the identity of the keys is scrambled, e.g., via hashing or other transformations. See discussion in Section 5.3.

is either a literal word or a wild-card pattern. Wild-card patterns are translated into regular expressions. In regex query mode, regex components of the query are directly interpreted as such. Search in the trie is recursive, with backtracking, starting with the first query-component. The search process iterates on the trie, generating candidates and filtering them against the query components.

For example, consider the query '*a_g?*_*' that starts with a literal component. If the first word, '*a*' is found among the unigrams, the set of its successors (code words) is pinpointed by a single address and can be loaded from a trie file to the compressed cache, and searched exhaustively. Each candidate code word (a byte-sequence) is converted into a word (via the reverse dictionary) and if the word matches the regular expression, we have a viable candidate. The node for the last byte of the candidate byte-sequence holds the offset for loading the block of next successors – that block is where candidates are sought for the next component of the query. If a candidate byte-sequence is found there, and its corresponding word is matched by the regular expression, we have a global match for the query. The frequency count of the n-gram is in the very last node of the retrieved sequence. To find a next global match, the system tries to find a next match for the last candidate component, '*'. Upon exhausting all candidates for the last word (given the current state of candidates for all preceding components of the query), the system backtracks to the handler of the previous component, '*g?**'. The block of (depth 3) successors that has been just exhausted is not needed anymore and may be unloaded from memory. If another candidate is found for the second component, a block of successors is loaded and searched. Eventually, the system backtracks to the first word/component and the search is exhausted. If the first component is not a literal but pattern, the recursion-backtracking process is repeated until backtracking reaches the root node. Handling of literal components is simpler. All literal components of a query are recoded into codeword byte-sequences. In any respective position on the trie, instead of searching a block of successors, only a single byte-sequence is sought. Complex regex or wild-card queries are served with very little memory load due to unloading of blocks (from Active Trie) upon backtracking.

When processing pattern-matching queries, in addition to returning all the matching n-grams from the database, TrendStream also automatically sums the counts (or frequencies) of all matching n-grams and outputs that value. There is also an alternate service mode. Instead of returning all the matching n-grams in one 'batch', a pattern-matching query can be served in Iterator Mode, in which matching n-grams are searched for, one by one, only when a *hasNext( )* request is issued by a calling application.

4 Empirical results

This section presents results of evaluations conducted on several different platforms. The list of platforms is presented in Appendix. The TrendStream system is implemented in Java (version 1.6), and can be used as an embedded library in applications, or run from scripts or command line. Evaluations presented here ran TrendStream from simple OS-level scripts (bash on Linux and .bat on Windows).

Table 3. *Original and filtered Google Web1T n-grams (unique n-gram types)*

Order	Original count	Retained count	Proportion of original
Unigrams	13,588,391	222,365	1.1%
Bigrams	314,843,401	92,661,145	29.4%
Trigrams	977,069,902	460,766,005	47.2%
4-grams	1,313,818,354	702,560,622	53.5%
5-grams	1,176,470,663	625,034,216	53.1%
Total	3,795,790,711	1,881,244,352	49.6%

4.1 Encoding: data and performance

We used data from two sources – Google Web1T and English Gigaword corpus (Graff and Cieri 2003). Encoding of the former demonstrates handling of pre-tabulated data, while the latter is an example of generating a large encoded collection of n-grams directly from streaming text.

Implementation of TrendStream provides configuration settings for filtering of punctuation, digits/numbers, and out-of-vocabulary words – depending on the settings, each of these can be either accepted, rejected, or encoded as a class (punctuation as '# PUNC', numbers as '# ', and unknown words as '# UNK'). For some practical text processing,[6] we generate n-gram models with controlled vocabulary of about 220 thousand word forms (a comprehensive coverage of English vocabulary[7]), punctuations accepted, but numbers converted to '# '. In addition, all accepted n-grams are converted to lowercase, and identical n-grams are merged (counts summed). With such filtering, we retained about half of the original Web1T data (see Table 3). The resulting TrendStream-encoded file stores about 1.8 billion unique n-grams.

During encoding, TrendStream makes use of the inherent updateability of its data format. For example, with the Web1T data, we encode the unigrams first, then the bigrams, trigrams, etc. Each time, the system simply updates more data into the compressed format, without recalculating any of the data that are already stored. There is just one single pass over the source data, reading directly from gzipped original tabulated files, with all filters applied on the fly. A single-threaded TrendStream process, on a Linux server (specification B), completed filtering and encoding of the Web1T collection in about 12 hours, utilizing one CPU and up to 16 GB of RAM. The optimized file size is 10.77 GB, with storage ratio of 6.15

[6] One of our uses for the language model is in the analysis of reading materials and student essays.

[7] Web1T has a vocabulary of 13.5 million 'words', but many of these are just numbers and dates, and web and email addresses. The need to filter this collection for some purposes is recognized in the literature (e.g., Islam and Inkpen, 2009b). For our purposes we also chose to exclude web and email addresses, common misspellings, and lots of other materials (e.g., dates), and focus only on a comprehensive coverage of English – by using a controlled vocabulary that includes all single words from WordNet 3.0 and all their inflected variants (multi-word terms were split and parts added as single words), plus English prepositions, pronouns, and wh-question words. This set was expanded to include additional vocabulary, as well as person and geographical names.

Table 4. *Statistics for n-grams collection generated from the Gigaword 2003 corpus*

Order	Unique n-grams	Total frequency (token count) for all n-grams of order n
Unigrams	222,364	1,762,201,096
Bigrams	32,534,152	1,683,823,411
Trigrams	202,483,396	1,609,558,543
4-grams	468,425,393	1,538,536,133
5-grams	674,240,506	1,470,29,5287
Total	1,377,905,811	

Note: Includes n-grams that span across sentences, uses controlled vocabulary.

bytes/n-gram. A version of the whole Google Web1T collection, without filters, but with uniform conversion to lower case, retains 3.35 billion distinct n-grams (out of 3.79 billion). Encoding of this data takes about 40 hours, utilizing one CPU and up to 25 GB of RAM. The resulting database file size is 20.4 GB, and the storage ratio is 6.53 bytes/n-gram.

The process of generating a language model from Gigaword texts was slightly different. TrendStream sequentially processed the Gigaword data directly from four very large text files (corresponding to the four sections of the Gigaword 2003 corpus, in total 4.5 GB uncompressed). Upon opening a text file, TrendStream simply defines a window of size *n* on the text, generates all n-grams of order 1 to *n* in the window, applies filters according to settings, and stores the n-grams in the Active Trie. It then continues by sliding the window until it reaches the end of file (resetting the window at end of sentence or at end of paragraph – depending on settings). Millions of new n-gram types (unique sequences) are generated and counts for millions of n-grams are updated continuously. During encoding, TrendStream automatically compresses data from the Active Trie to the Compressed Cache and to disk, and it also moves data in the other direction – when updates are needed for stored n-grams that are no longer in the Active Trie. The single-threaded TrendStream process on a non-dedicated Linux server (spec. B) completes generation (with filtering) and optimized encoding of n-grams (orders 1–5) for the Gigaword corpus in about 24 hours using one CPU and up to 20 GB of RAM. The resulting optimized file size is 7.3 GB, with the storage ratio of 5.68 bytes/n-gram. Statistics for the Gigaword n-grams collection are presented in Table 4. Unlike the Web1T data, this dataset includes many n-grams with very low counts (e.g., 1, 2).[8] We henceforth refer to these databases, with filtered data, as Web1T.f and Gigaword.f.

[8] Note that our version of Gigaword-2003 n-grams collection includes n-grams (n = 1–5) that span across sentence boundaries (thus, resulting in about 1.3 billion n-grams), while the standard approach is to include only n-grams within sentences, which leads to a much smaller total amount (e.g., 415 million n-grams; Germann et al. 2009).

4.2 N-gram retrieval performance

The TrendStream n-gram retrieval performance was tested with a benchmark task. We used a collection of texts totaling about 10 million words.[9] Two massive retrieval tasks were defined – one for bigrams and another for 5-grams. On each task, the system scans text files with a window of two or five words (with punctuation marks treated as elements), and attempts to retrieve n-gram counts from the database. In this task, repeated queries come about only as they naturally occur in the text. Evaluation runs were conducted with the two filtered encoded language models described in the previous section, on three platforms differing in their computational power. Note that deploying of the language models was extremely simple – we just copied the TrendStream software (less than 1 MB) and the compiled n-grams file (7.3 or 10.77 GB).

Memory use and search times were measured for two different load options – on-demand loading and whole-model loading.[10] Since the on-demand mode heavily uses disk access, it was tested in two conditions – cold start (with empty OS-caches, all data read from disk) and warm start (some OS-caches are reused). For each condition, timing and memory values were averaged across three separate runs. Results are given in Table 5.

In on-demand loading mode, disk access time is a notable factor – the system is rather slow in the cold-start runs (however, time to first response is always under 1 second). As the operating system caching kicks in, the TrendStream performance accelerates and reaches levels that are quite suitable for some intensive text-processing tasks. We get thousands and even hundreds of thousands of queries served per second.

Performance in the full-loading mode is about two to three times faster than on-demand loading with warm cache. However, there is a huge difference in memory consumption. Full loading consumes several gigabytes of RAM. With on-demand loading, TrendStream performs the same task a little slower, but with dramatically less memory. For bi-grams retrieval, memory use averages are below 150 MB and peak usage values are all below 220 MB. For the 5-grams, averages are below 500 MB, with peak loads all below 700 MB. Notably, on-demand loading is not just lazy memory-mapping from disk – the TrendStream memory manager actively releases parts of the trie from memory, which is why average memory values are much lower than peak values.

We also conducted an evaluation with a database that includes the full (unfiltered) set of data from Google Web1T (the compiled TrendStream file of 20.4 GB). Results are presented in Table 6. With a larger model, performance is slower (compared with Table 5) because search-space is larger, but the speed is still in thousands of queries per second.

[9] This collection consists of about 40% newswire text (from the WSJ section of the Penn Treebank), 40% encyclopedia articles (Encarta), and 20% student essays (responses on various writing prompts for the Graduate Record Examinations (GRE) test).

[10] The same database loads in different modes on different occasions, controlled by an option switch.

Table 5. *TrendStream retrieval performance for 2- and 5-grams, with two different models on several different platforms*

Platform	Data source of n-gram database	2-grams retrieval task			5-grams retrieval task		
		Average rate queries/ms	RAM use (MB)		Average rate queries/ms	RAM use (MB)	
			Average	Peak		Average	Peak
Full model loading to RAM (*memory use plus 7.3 or 10.77 GB)							
(a) Linux server, high-end	GigaWord.f	1200	83*	161*	247	336*	498*
	Web1T.f	1038	110*	210*	241	433*	651*
(b) Linux server, mid-range	GigaWord.f	918	80*	124*	168	336*	491*
	Web1T.f	719	105*	154*	129	*432**	*640**
(c) Linux server, low-end	GigaWord.f	358	33*	116*	57	105*	238*
	Web1T.f	312	42*	101*	37	112*	261*
On-demand loading – warm-start							
(a) Linux server, high-end	GigaWord.f	668	92	181	92	344	516
	Web1T.f	599	123	219	85	442	650
(b) Linux server, mid-range	GigaWord.f	482	86	144	53	342	503
	Web1T.f	377	113	163	47	440	656
(c) Linux server, low end	GigaWord.f	165	67	175	25	341	500
	Web1T.f	108	113	163	14	442	662
On-demand loading – cold-start							
(a) Linux server, high-end	GigaWord.f	199	92	181	61	344	516
	Web1T.f	156	123	218	50	442	651
(b) Linux server, mid-range	GigaWord.f	45	87	145	32	340	500
	Web1T.f	37	114	162	27	441	657
(c) Linux server, low-end	GigaWord.f	17	84	136	8	343	505
	Web1T.f	15	113	163	7	443	659

Notes: Platform specifications (a)–(c) are listed in the Appendix. Timing values reflect pure search times (excluding text files I/O). Each memory value is sum of Active Trie and Compressed Cache, except for full loading mode where values reflect Active Trie only and Compressed Cache is the size of the database file.

Table 6. *TrendStream retrieval performance for 2- and 5-grams with all Google Web1T data on a high-end Linux server*

Platform	2-grams retrieval task			5-grams retrieval task		
	Average rate queries/ms	RAM use (MB)		Average rate queries/ms	RAM use (MB)	
		Average	Peak		Average	Peak
The 20-GB model fully loaded in RAM	310	107+*	169+*	134	368+*	559+*
On-demand loading – warm-start	151	495	579	45	1120	1200
On-demand loading – cold-start	140	559	661	39	946	1167

Note: In full-load mode, RAM use is 20.4 GB + the amount indicated.

Full loading and on-demand loading are two extreme cases – the first optimized for speed, and the second optimized for minimizing memory consumption. TrendStream also provides a flexible mode of loading via a parameter setting that specifies how much memory should be used for mapping. For example, working with a 10-GB model file, we can specify to memory map 3 GB (or any other amount) and the rest is handled by on-demand loading.

With on-demand loading, TrendStream can run on platforms that do not have enough RAM for a whole model. This flexibility extends to using the system even on the modest desktop and laptop platforms. On modest platforms, n-gram models would not be used to analyze massive amounts of text, but they can be useful for research, and for NLP tasks that need quick processing of relatively small amount of text. One such task is contextual spelling correction. For example, Microsoft Office 2007 uses a pruned trigram language model that is tightly compressed into a few megabytes (Church, Hart and Jianfeng 2007). Here we demonstrate running very large, multi-gigabyte 5-grams language models on the modest hardware. We use the same database files described above (Gigaword.f and Web1T.f), and the set of platforms includes a desktop and laptop, each with just 4-GB RAM (specifications D and E). For a small-scale performance evaluation task, we selected a single text[11] with 103,666 tokens (83,592-word tokens and the rest are punctuation tokens). Separate evaluations were run for retrieval of two-grams and 5-grams: on each run, the text was processed with a two- or five-word sliding window. Timing values were averaged across three separate runs in each condition. Results are presented in Table 7. Not surprisingly, there is a marked difference in performance between cold and warm starts, indicating that disk access is the most significant source of delay. Even in cold start, on our slowest platform, the system serves between 460 bigrams and about 120 5-grams per second, which is suitable for immediate processing of short documents.

[11] *The Children's Book of Christmas Stories*, from Project Gutenberg, available at http://www.gutenberg.org/etext/5061

Table 7. *TrendStream retrieval performance for 2- and 5-grams, for one test file with two different models on several different platforms*

Platform	Data source of n-gram database	2-grams retrieval task			5-grams retrieval task		
		Total search time (ms)	Average rate 2-grams/ms	RAM use (MB)	Total search time (ms)	Average rate 5-grams/ms	RAM use (MB)
On-demand loading – warm-start							
(a) Linux server, high-end	GigaWord.f	740	140.08	62	3514	29.50	257
	Web1T.f	1121	92.47	75	4752	21.81	333
(b) Linux server, mid-range	GigaWord.f	1029	100.74	63	3957	26.19	257
	Web1T.f	1156	89.67	79	4981	20.81	327
(c) Linux server, low-end	GigaWord.f	1691	61.30	63	6597	15.71	250
	Web1T.f	2085	49.72	77	8967	11.56	327
(d) Windows 7 laptop (64-bit)	GigaWord.f	3072	33.75	58	13,444	7.71	248
	Web1T.f	3324	31.19	78	17,474	5.93	325
(e) Windows 7 desktop (32-bit)	GigaWord.f	4297	24.12	42	22,872	4.53	154
	Web1T.f	4819	21.51	52	25,993	3.98	206
On-demand loading – cold-start							
(a). Linux server, high-end	GigaWord.f	74,971	1.38	63	3,33,860	0.31	256
	Web1T.f	76,369	1.36	78	3,72,763	0.28	333
(b) Linux server, mid-range	GigaWord.f	74,530	1.39	63	4,01,801	0.26	256
	Web1T.f	76,788	1.35	79	3,69,900	0.28	317
(c) Linux server, low end	GigaWord.f	1,32,873	0.78	63	4,61,298	0.23	250
	Web1T.f	1,26,908	0.82	78	5,18,310	0.20	327
(d) Windows 7 laptop (64-bit)	GigaWord.f	2,13,316	0.49	58	8,09,433	0.13	248
	Web1T.f	22,3574	0.46	78	8,74,631	0.12	325
(e) Windows 7 desktop (32-bit)	GigaWord.f	16,9874	0.62	42	6,15,197	0.17	161
	Web1T.f	1,70,670	0.60	52	6,56,525	0.16	206

Notes: Platform specifications (a)–(e) are listed in the Appendix. Average rate values reflect pure search times (not including text files I/O). Each memory value is sum of Active Trie and Compressed Cache.
For 2-gram retrieval task, the number of queries is 103,665, of them 93.3% are found in the Web1T.f model, 91.9% are found in the GigaWord.f model.
For 5-gram retrieval task, the number of queries is 103,662, of them 25.5% are found in the Web1T.f model, 14.3% are found in the GigaWord.f model.

On the modest platforms, with warm start the performance is thousands of queries per second, which is adequate for some serious NLP processing.[12]

4.3 Performance with pattern matching queries

The TrendStream performance for pattern-matching queries is illustrated in Table 8. Six sample wildcard queries and three sample regex queries were run against the Web1T.f database file on three different platforms. On the Linux server (spec. B, 32-GB RAM), we ran queries in two different memory modes – one with full model loading to RAM, and another with on-demand loading. The Windows laptop has just 4-GB RAM, so only on-demand loading was used. All on-demand loading queries were run with warm OS-cache. Timing and memory-use values were averaged across three separate runs in each condition.

Query 1, '*to_*_b*d*', returns 19,466 trigrams, among them '*to address beyond*' and '*to international boulevard*'. Query 2, '*a_????_b*d_r*_*', returns 165 5-grams, among them '*a rock band rather than*' and '*a very broad representation of*'. Query 1 is much shorter, but it takes more time to process. This illustrates how the system uses query constraints to limit search. Both queries have a highly frequent first word (*to*/*a*), with large sets of successors for next word (depth 2). With query 1, for each candidate at depth 2 the system proceeds to a deeper level, seeking a third word. For query 2, it proceeds to a deeper level only for four-lettered candidates at depth 2. Notably, except for the 'accept any' ('*' or '.+'), all regular expressions are used as filters and help reduce the search space.

It might be expected that a fronted wildcard query would be quite difficult and time-consuming to process on the TrendStream data structure. However, this is not the case. We tested two fronted wildcard queries. Query 3, '**d*', retrieves unigrams, and Query 4, '* *was*', retrieves bigrams. Query 3 scans over all the unigrams in the model. Query 4 also scans on all the unigrams, and for each one it checks the successors at depth 2. Performance on these simple fronted wildcard queries is quite fast, even faster than performance for the more constrained queries 1 and 2. To put the system to a more demanding test, we ran Query 5: '**t* **t*'. Here it must scan all unigrams, and for each accepted unigram it needs to perform a wildcard search at level 2. The system retrieves about half a million matching bigrams in 21 seconds (fastest, 85 seconds on the slowest machine). We then tried a more demanding Query 6, '**t* **t a*', which returns 15,782 trigrams. This query has to visit all the n-grams that '**t* **t*' visited, and for each of them steps into level 3, looking for next word 'a'. Query 6 completes in 21 seconds (88 seconds on the slowest machine). Note that this is done without any additional indexing, inverted or rotated structures, and with very modest use of RAM.

[12] For example, an application with the embedded TrendStream library performs n-gram analysis for thousands of student essays – within several minutes, running on the 4-GB Windows laptop. Our analyses require tens of thousands of queries per short essay, which is admittedly unlike SMT processing, where a short text may call for millions of n-gram queries.

Table 8. *TrendStream retrieval performance for some wildcard and regex queries with Web1T.f database*

#	Query	n-gram order	n-grams retrieved	10.77-GB model fully in RAM		On-demand loading					
				Linux server, mid-range (b)		Linux server, high-end (a)		Linux server, mid-range (b)		Windows 7 laptop (d)	
				Time (sec)	RAM (MB)*	Time (sec)	RAM (MB)	Time (sec)	RAM (MB)	Time (sec)	RAM (MB)
1.	to * b*d	3	19,466	14.27	329+	14.14	626	27.68	381	54.61	327
2.	a ???? b*d r* *	5	165	2.08	129+	2.83	180	4.66	134	8.92	136
3.	*d	1	14,755	0.81	68+	1.22	163	1.53	69	2.04	82
4.	* was	2	110,301	1.31	228+	3.19	232	4.57	231	12.38	238
5.	*t *t	2	491,508	21.65	397+	20.91	306	41.71	410	85.08	261
6.	*t *t a	3	15,782	21.08	371+	21.25	298	39.29	374	88.35	322
7.	(?:p.+|work|sav|check) (?:ed|ing) for any|money|free	3	275	0.84	77+	1.05	118	1.52	83	2.69	64
8.	was (?:con.+|mark|delet) (?:ed|ing) .+ [aeiou].+ion	4	144	0.75	65+	0.99	76	1.43	73	2.19	60
9.	.{3,5} (?:im|ex|com)press [^b].+	3	15,063	57.01	*377*	70.89	305	138.96	330	380.59	312

With full regular expression syntax, TrendStream allows to formulate even more sophisticated conditions for n-gram retrieval: a regular expression (or a literal string) for every element 'word' of the query (but not across words). Sample Query 7 has regex first and third elements, and a very frequent second word. It returns 275 trigrams, among them '*parking for free*' and '*worked for money*'. Query 8 starts with a frequent word ('was'), and continues with three regular expressions, which require many 'check the whole next level' steps. Results include n-grams such as '*was marked by innovation*' and '*was considering this option*'. Query 9 is quite complex – it has regular expressions for each of its three elements, and accepts any first word that is three- to five-letter long. Thus, it requires a deep and extensive scan into the database. It retrieves 15,063 trigrams, such as '*polar express caboose*', '*can impress audiences*', and '*and compress video*'.

Overall, performance times are rather fast, considering that the system searches in a database of 1.8 billion n-grams. In on-demand loading mode, performance on the slowest machine in our test set is only about four times slower than on the high-end server. Memory use for pattern matching queries is remarkably low. This is achieved by two factors. First, the search is automatically guided and loads (from disk) only those chunks of data that are potentially useful. Second, upon backtracking, the recursive search automatically releases memory by unloading used portions of the Active Trie – those from which it backtracks. Upon completion of the search, the system can either release all of the memory or keep some parts of the Active Trie for serving another query. For queries that are expected to return thousands of matches, there is some speed advantage when the n-gram model is fully loaded in memory. This advantage can also be effective when many pattern-matching queries need to be served (e.g., in a batch).

We have also run pattern-matching queries with the complete Web1T database (20.4-GB file). Query '.+\.*com*' asks how many .com web addresses are among the unigrams in Web1T. With full model loaded in memory (Linux server, spec. B), the query is processed in 16 seconds (iterating over several million unigrams and applying pattern matching to each). Queries '* *was*' and '* *is*' are processed in 25 seconds each, '*sit|sits|sat|sitting* .+ .{*1*} *chair*' takes 10 seconds, and '.+*ly good fun*' takes 15 seconds.

5 Discussion

5.1 Comparison with other systems

While an extensive comparison of TrendStream with other systems is beyond the scope of this paper, we have gathered some data based on the published results (Table 9). The comparison is far from exact, given different sizes of text corpora, differences in filtering/pruning policies, smoothing, quantization, and computing on platforms with different hardware specifications. Current implementation of TrendStream supports only 'Stupid-backoff' smoothing (Brants *et al.* 2007), and allows setting any value for the backoff parameter.

Most of the specialized LM toolkits are designed for top speed, typically requiring to maintain a whole model in memory. Their most prominent utilization is in

Table 9. *Rough comparison of various n-gram storage systems*

System	Architecture		Google Web1T		Gigaword or similar			Pattern matching queries
			Bytes/ n-gram	Database size (GB)	Bytes/ n-gram	2-grams queries/ms	5-grams queries/ms	
MySQL 5.0	RDBMS	*		260 $_{CF}$		0.5$_{GH}$		Possible
Web1T5-Easy	RDBMS + Perl scripts			180 $_{E}$				Yes
B+ indexer	Index over flat files			86 $_{CM}$			1.0 $_{CM}$	
	+ more indexes for rotated n-grams			420 $_{CM}$			1.0 $_{CM}$	Limited
SRILM	Default (trie with hash tables)				42.2 $_{GJL,PK}$		750 $_{KH}$	No
	Compact (trie with sorted arrays)			~116 $_{PK}$	33.6 $_{GH,PK}$	627 $_{GH}$	238 $_{KH}$	
IRSTLM	Sorted trie				14.2$_{GJL,GH}$	368 $_{KH}$		No
	Same with 8-bit quantized data	*			9.10 $_{GJLGH}$	422 $_{GH}$	402 $_{KH}$	
	Inverted sorted trie					426 $_{KH}$		
TPT	Trie with variable length compression		4.52 $_{GJL}$	16 $_{GJL}$	7.50 $_{GJL}$		357 $_{KH}$	No
TrendStream	Trie with variable length compression		**6.53**	*20.4*	*5.68 to 6.15*	*1200*	*247*	**Yes**
RandLM [1]	12-bit fingerprints		7.53 $_{GH}$	26.63 $_{GH}$	6.00 $_{GH}$			No
	12-bit fingerprints and 8-bit data quantization	*	3.08 $_{GH}$	~10.9	3.08 $_{GH}$			
	8-bit fingerprints		6.91 $_{GH}$	~24.43	5.38 $_{GH}$			
	8-bit fingerprints and 8-bit data quantization	*	2.46 $_{GH}$	~8.69	2.46 $_{GH}$	33.8 $_{GH}$	56 $_{KH}$	
MPHR[2]	S-MPHR, with 12bit fingerprints		4.26 $_{GH}$	15.05 $_{GH}$	3.76 $_{GH}$			No
	S-MPHR and payload data quantized 8b	*	2.76 $_{GH}$		2.76 $_{GH}$			
	C-MPHR: 12bit fingerprints		3.40 $_{GH}$		2.19 $_{GH}$			
	C-MPHR, with 8-bit data quantization	*	2.09 $_{GH}$		2.09 $_{GH}$	507 $_{GH}$	~607$_{KH}$	
	T-MPHR, with 12bit fingerprints		2.97 $_{GH}$	~10.5	2.16 $_{GH}$			
	T-MPHR, with 8-bit data quantization	*	1.91 $_{GH}$	~8.74	1.91 $_{GH}$			

Table 9. *Continued*

System	Architecture	Google Web1T		Gigaword or similar			Pattern matching queries
		Bytes/ n-gram	Database size (GB)	Bytes/ n-gram	2-grams queries/ms	5-grams queries/ms	
BerkeleyLM	SORTED: trie with sorted arrays	10.5 PK	37 PK				No
	SORTED, with implicit encoding	6.5 PK	23 PK	8.5 PK		714 PK	
	HASH: trie with hash tables	15 PK	53 PK				
	HASH, with implicit encoding	9.1 PK	32 PK	11.6 PK		2000 PK 776 KH	
	COMPRESSED: trie + variable length encoding + block compression	2.9 PK	10.2 PK	5.9 PK		108 PK 126 KH	
KenLM	PROBING:						
	hash table for each n-gram order N					1818 KH	No
	TRIE: inverted trie with sorted arrays					1139 KH	
	TRIE, with 8-bit data quantization *					1127 KH	

Notes: RandLM uses randomized hashing based on the Bloomier filter (Talbot and Brants, 2008).
MPHR systems use randomized hashing via minimal perfect hashing functions (Guthrie and Hepple, 2010a, 2010b).
*Lossy compression; ∼an estimated value.
CF – Carlson and Fette (2007); CM – Ceylan and Mihalcea (2011); E – Evert (2010); GJL – Germann*et al.* (2009); GH – Guthrie and Hepple (2010b); KH – Heafield (2011); PK – Pauls and Klein (2011).

statistical machine translation systems, where n-gram query processing loads are extremely high, and all queries are for literal n-grams. On the other hand, systems like Web1T5-Easy (Evert 2010) and B+ Indexer (Ceylan and Mihalcea 2011) are mostly disk-based (and thus much slower), and oriented for research purposes and less intensive tasks, but with more features such as wildcard queries. TrendStream is a unique system as it is oriented for both sides – it can be fully loaded in memory for fast intensive processing and can work in low memory mode when needed. Notably, low memory mode is needed in many scenarios, for example, (a) when many applications and services run on the same platform (and greedy memory consumption is not 'allowed'), (b) for tasks that do not require ultra-fast intensive processing, and (c) on platforms that simply do not have enough memory. Currently, TrendStream is the only flexible system that can work in both modes, and can even switch dynamically from one mode to another.

Several authors mention file size and compression ratios for the full Google Web1T corpus. Some of the best compression results are obtained by randomized hashing techniques, although a recent trie-based system matches even these results (Pauls and Klein 2011). On this 'metric', TrendStream is somewhere in the middle – its encoding of the full Web1T data is twice as large as the best-compression systems, but it is still better than many other systems.

Most of the published performance timing comparisons use n-gram models based on data from the Gigaword corpus or text collections of roughly comparable size. Some evaluations run queries for specific n-gram order (2 or 5), while some report results on perplexity computation or similar task, where queries for n-grams of various orders may be mixed. TrendStream compares quite well – it is not among the fastest systems, but not far behind.

5.2 Some uses for pattern-matching n-gram queries

One interesting potential for pattern-matching queries is in spelling correction. Carlson and Fette (2007) and Islam and Inkpen (2009b, 2009c) demonstrated use of Web1T for spelling correction. In these studies, the n-gram frequency data from Google Web1T is used to filter and re-rank correction candidates, but the candidate corrections are obtained from elsewhere (typically via string similarity algorithms). With pattern-matching queries, we can obtain candidates directly from a language model. For example, given a misspelled word 'm1' in context 'w1 w2 m1 w3 w4', a list of potential correction candidates, with contextual probabilities, can be obtained directly via a query like '*w1 w2* * *w3 w4*'.

The capability to perform pattern-matching queries gives rise to another interesting functionality – computing distributional similarity for pairs of words, based on information in very large n-gram databases. When distributional similarity is computed using vectors of words' immediate predecessors and successors, it captures some aspects of both semantic and syntactic information. Obtaining such vectors with TrendStream is simple: for two words, w1 and w2, successor vectors are obtained via queries '*w1* *' and '*w2* *'. Predecessor vectors can be obtained via queries '* *w1*' and '* *w2*' (in practice, the system uses a single query based on a variant of '.+ *w1*|*w2*').

TrendStream provides options for calculating vector similarity with just right, left, or combined right and left vectors, and output vector sizes. Supported similarity measures include cosine, Dice coefficient, Jaccard coefficient, etc. (Manning and Schütze 1999). For example, cosine similarity of (desk, table) using combined vectors is 0.20 (Web1T.f) or 0.29 (Gigaword.f). For a simple vector similarity application, we compiled a database file that includes only unigrams and bigrams from the Web1T collection (filtering with controlled vocabulary). The resulting model has 928 million n-grams and file size is just 462 MB. Thus, it can be held fully in memory even on very modest hardware. With this database, comparing high-frequency words ('was', 'is') produces a combined vector of size 165,191. Cosine value is 0.816, and Dice value is 0.848. Vector similarity is computed in 1–3 seconds (depending on platform). For less frequent words (i.e., smaller vectors), performance is much faster, typically about 0.5–1 seconds.

5.3 Enumerating keys and merging models

The TrendStream ability to serve pattern-matching queries stems from an architectural design feature – the capability to enumerate all text keys in the data structure. In principle, a simple trie (prefix tree) should be able to enumerate all the keys that are stored in it, simply by iterating over all of them. Each valid key has some data associated with it, and that are sufficient to distinguish valid keys from any other substrings. Most modern LM toolkits recode words into numeric IDs (or hash values) for better compression and faster retrieval. If during encoding each n-gram is transformed word-by-word into some coded representation, a reverse dictionary (from numeric IDs to words) can help recover textual identity of the stored n-grams. Such dictionary is rather small, even for a large vocabulary. The capability to recover original n-gram text is lost if n-grams are transformed (e.g., hashed) as whole strings, rather than word-by-word.

TrendStream can enumerate all valid stored keys and the associated data. Its architecture also supports unlimited insertion of new keys and data in an existing compiled database. Together, these two features give rise to an interesting new capability – they allow merging two compiled encoded n-gram models into one. The process is very simple – we iterate on all valid keys in one encoded file, the source, and send the keys and data for insertion into another encoded file, the recipient. Note that the source and the recipient might have totally different coding (e.g., they assign different numeric IDs for same words), and that is why recovering textual identity of the n-grams is important in this case. The receiver, upon reception of new key and data, simply checks if such n-gram is already stored. If it is stored, it just updates count/frequency, and if it is not stored, a new n-gram is inserted with associated data. In our experiments, we merged a TrendStream database file of about 1 GB with another TrendStream database file of 7-GB size (more than a billion n-grams, of orders 1–5). Using the larger file as recipient, the merging process takes about 60 minutes (on Linux server, spec. A). The merging capability is rather unique among various LM toolkits. It may have some interesting uses, both for static and streaming scenarios. One potential is for time-based n-gram generation

and encoding from textual streams – encode data per day/week/etc., then merge into a larger database file. Another potential is for organizing libraries of n-grams from different large text corpora. For example, one can have a n-gram database trained on Gigaword, another database trained on Wikipedia, and a merged one – which is created without retraining. Another use: parallelize the process of producing a language model – build and encode partial models on several machines, then merge the encoded files.

5.4 Utilization

At the Educational Testing Service, TrendStream is already routinely used as an embedded library in text-processing applications. For example, TrendStream is embedded in automatic spelling correction system that uses n-grams for contextual re-ranking of correction alternatives (manuscript in preparation). It is also used in a system that identifies atypical word combinations in essays written on TOEFL examinations (Futagi 2010), and in many internal research projects. In most of these cases, an application runs either on a low-end machine that cannot load a multi-gigabyte n-gram model in RAM (e.g., laptop or desktop), or on a server – concurrently with many other modules and memory-hungry applications. In such scenarios, the capability of TrendStream to run with minimal memory footprint is a direct and obvious benefit. With simple portability and flexible memory footprint, the TrendStream library also simplifies porting applications across different platforms and even into a cloud computing environment. There is also a graphical user interface (GUI) extension to the TrendStream system – a standalone desktop application that supports interactive n-gram queries with multiple listings, tabbed interface, sorting, and copy/print/export facilities.

5.5 Conclusions

This paper presented a versatile architecture for storage and retrieval of very large word n-gram datasets. It belongs to the family of trie-based architectures, but differs in several respects. While most trie-based systems are word-oriented, TrendStream adopts character- and byte-level granularity for trie nodes. Instead of hash-tables or sorted arrays, a hybrid Ternary Search Trie data structure is used, both in memory and for storage in serialized form. A new variant of lossless variable-length compression is used for both the indexing structure (n-grams) and the associated frequency counts. This scheme achieves competitive compression rates and supports fast searches. Due to the byte-level granularity and ternary organization of the trie, the search algorithm automatically guides loading and decompression of the stored data.

The TrendStream toolkit provides fast searches, flexible memory requirements, updateable language models and is highly portable. It also supports pattern-matching queries without any additional indexing. Such versatile architecture opens the road for wide adoption of very large n-gram databases on a variety of platforms, from high-end servers to the modest desktops and laptops.

Acknowledgments

Many thanks to Joel Tetreault and Nitin Madnani for useful discussions, to Yoko Futagi, Lei Chen, and Alla Rozovskaya for detailed comments on an earlier draft. This manuscript has also greatly benefited from the comments of anonymous reviewers.

References

Bentley, J. L., and Sedgewick, R. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'97), New Orleans, LA, USA, pp. 360–9. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Bentley, J. L., and Sedgewick, R. 1998. Ternary search trees. *Dr Dobb's Journal*, April 01, http://drdobbs.com/windows/184410528. Accessed 8 April 2011.

Bergsma, S., Lin D., and Goebel, R. 2009. Web-scale n-gram models for lexical disambiguation. In *Proceedings of 2009 International Joint Conference on Artificial Intelligence (IJCAI 2009)*, Pasadena, CA, USA, pp. 1507–12. Palo Alto, CA: AAAI Press.

Bloom, B. H. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* **13**(7): 422–6.

Brants, T., and Franz, A. 2006. *Web 1T 5-gram Version 1*. Philadelphia, PA, USA: Linguistic Data Consortium. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2006T13

Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. 2007. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing (EMNLP-CoNLL 2007)*, Prague, Czech Republic, pp. 858–67. Stroudsburg, PA: Association for Computational Linguistics.

Carlson, A., and Fette, I. 2007. Memory-based context-sensitive spelling correction at web scale. In *Proceedings of the Sixth International Conference on Machine Learning and Applications (ICMLA '07)*, Cincinnati, OH, USA, pp. 166–71. Los Alamitos, CA, USA: IEEE Computer Society Press.

Ceylan, H., and Mihalcea, R. 2011. An efficient indexer for large n-gram corpora. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (ACL-HLT 2011), System Demonstrations*, Portland, OR, USA, pp. 103–8. Stroudsburg, PA: Association for Computational Linguistics.

Chazelle, B., Kilian, J., Rubinfeld R., and Tal, A. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, New Orleans, LA, USA, pp. 30–9. Philadelphia, PA: Society for Industrial and Applied Mathematics.

Church, K., Hart, T., and Jianfeng, G. 2007, Compressing trigram language models with Golomb coding. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, Prague, Czech Republic, pp. 199–207. Stroudsburg, PA: Association for Computational Linguistics.

Emami, A., Papineni, K., and Sorensen, J. 2007. Large-scale distributed language modeling. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP-2007*, Honolulu, HI, USA, vol. 4, pp. 37–40. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Evert, S. 2010. Google Web 1T 5-grams made easy (but not for the computer). In *Proceedings of the NAACL HLT 2010 Sixth Web as Corpus Workshop* (WAC-6), Los Angeles, CA, USA, pp. 32–40. Stroudsburg, PA: Association for Computational Linguistics.

Federico, M., and Cettolo, M. 2007. Efficient handling of n-gram language models for statistical machine translation. In *Proceedings of the ACL 2007 Workshop on Statistical*

Machine Translation, Prague, Czech Republic, pp. 88–95. Stroudsburg, PA: Association for Computational Linguistics.

Fredkin, E. 1960. Trie memory. *Communications of the ACM* **3**(9): 490–9.

Futagi, Y. 2010. The effects of learner errors on the development of a collocation detection tool. In *Proceedings of the Fourth Workshop on Analytics for Noisy Unstructured Text Data (AND '10)*, Toronto, Canada, pp. 27–34. New York, NY: Association for Computing Machinery.

Germann, U., Joanis, E., and Larkin, S. 2009. Tightly packed tries: how to fit large models into memory, and make them load fast, too. In *Proceedings of the NAACL HLT Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, Boulder, CO, USA, pp. 31–9. Stroudsburg, PA: Association for Computational Linguistics.

Giuliano, C. 2007. jWeb1T: a library for searching the Web 1T 5-gram corpus. Software available at http://hlt.fbk.eu/en/technology/jWeb1t. Accessed 8 April 2011.

Graff, D., and Cieri, C. 2003. *English Gigaword*. Philadelphia, PA, USA: Linguistic Data Consortium. http://www.ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2003T05

Guthrie, D., and Hepple, M. 2010a. Minimal perfect hash rank: compact storage of large n-gram language models. In *Proceedings of SIGIR 2010 Web N-gram Workshop*, Geneva, Switzerland, pp. 21–9. New York, NY: Association for Computing Machinery.

Guthrie, D., and Hepple, M. 2010b. Storing the Web in memory: space efficient language models with constant time retrieval. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing (EMNLP-2010),* Boston, MA, USA, pp. 262–72. Stroudsburg, PA: Association for Computational Linguistics.

Harb, B., Chelba, C., Dean, J., and Ghemawat S. 2009. Back-off language model compression. In *Proceedings of 10th Annual Conference of the International Speech Communication Association (Interspeech 2009)*, Brighton, UK, pp. 325–55. International Speech Communication Association, ISCA archive http://www.isca-speech.org/archive/interspeech_2009. Accessed 8 April 2011.

Hawker, T., Gardiner, M., and Bennetts, A. 2007. Practical queries of a massive n-gram database. In *Proceedings of the Australasian Language Technology Workshop 2007* (ALTW 2007), Melbourne, Australia, pp. 40–8. Australasian Language Technology Association.

Heafield, K. 2011. KenLM: faster and smaller language model queries. In *Proceedings of the 6th Workshop on Statistical Machine Translation*, pp. 187–97, Edinburgh, Scotland, UK. Stroudsburg, PA: Association for Computational Linguistics.

Islam, A., and Inkpen, D. 2009a. Managing the Google Web 1T 5-gram data set. In *Proceedings of International Conference on Natural Language Processing and Knowledge Engineering (NLP-KE 2009)*, Dalian, China, pp. 1–5. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Islam, A., and Inkpen, D. 2009b. Real-word spelling correction using Google Web 1T n-gram data set. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009)*, Hong Kong, pp. 1689–92. New York, NY: Association for Computing Machinery.

Islam, A., and Inkpen, D. 2009c. Real-word spelling correction using Google Web 1T n-gram with backoff. In *Proceedings of the IEEE International Conference on Natural Language Processing and Knowledge Engineering (IEEE NLP-KE'09)*, Dalian, China, pp. 1–8. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Lapata, M., and Keller, F. 2005. Web-based models for natural language processing. *ACM Transactions on Speech and Language Processing* **2**(1): 1–31.

Levenberg, A., and Osborne, M. 2009. Stream-based randomised language models for SMT. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing (EMNLP-2009)*, Singapore, pp. 756–64. Stroudsburg, PA: Association for Computational Linguistics.

Manning, C., and Schütze, H. 1999. *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Pauls, A., and Klein, D. 2011. Faster and smaller n-gram language models. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011)*, Portland, OR, USA, pp. 258–67. Stroudsburg, PA: Association for Computational Linguistics.

Raj, B., and Whittaker, E. W. D. 2003. Lossless compression of language model structure and word identifiers. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'03)*, Hong Kong. USA, vol. 1, pp. 388–99. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Ravishankar, M. 1996. *Efficient Algorithms for Speech Recognition*. PhD thesis, Technical Report. CMU-CS-96-143, Carnegie Mellon University, Pittsburgh, PA, USA.

Sekine, S. 2008. Linguistic knowledge discovery tool: very large n-gram database search with arbitrary wildcards. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, Manchester, UK, pp. 181–4. Stroudsburg, PA: Association for Computational Linguistics.

Sekine, S., and Dalwani, K., 2010. N-gram search engine with patterns combining token, POS, chunk and NE information. In *Proceedings of Language Resource and Evaluation Conference (LREC-2010)*, Valletta, Malta, pp. 2682–6. Paris, France: European Language Resources Association.

Stolcke, A. 2002. SRILM – An extensible language modeling toolkit. In *Proceedings of 7th International Conference on Spoken Language Processing* (ICSLP2002 – INTERSPEECH 2002), Denver, CO, USA, pp. 901–4. International Speech Communication Association, ISCA archive, http://www.isca-speech.org/archive/icslp02

Talbot, D. 2009. Succinct approximate counting of skewed data. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI 2009)*, Pasadena, CA, USA, pp. 1243–8. Palo Alto, CA: AAAI Press.

Talbot, D., and Brants, T. 2008. Randomized language models via perfect hash functions. In *Proceedings of 46th Annual Meeting of the Association for Computational Linguistics and Human Language Technology Conference (ACL-08: HLT)*, Columbus, OH, USA, pp. 505–13. Stroudsburg, PA: Association for Computational Linguistics.

Talbot, D., and Osborne, M. 2007a. Randomised language modelling for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics (ACL 2007)*, Prague, Czech Republic, pp. 512–19. Stroudsburg, PA: Association for Computational Linguistics.

Talbot, D., and Osborne, M. 2007b. Smoothed Bloom filter language models: tera-scale LMs on the cheap. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, Prague, Czech Republic, pp. 468–76. Stroudsburg, PA: Association for Computational Linguistics.

Van Durme, B., and Lall, A. (2009). Probabilistic counting with randomized storage. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI 2009)*, Pasadena, CA, USA, pp. 1574–9. Palo Alto, CA: AAAI Press.

Wang, K., and Li, X. 2009. Efficacy of a constantly adaptive language modeling technique for web-scale applications. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing* (ICASSP'2009), Taipei, Taiwan, pp. 4733–6. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Watanabe, T., Tsukada, H., and Isozaki, H. 2009. A succinct n-gram language model. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing (ACL-IJCNLP 2009)*, Short Papers, Suntec, Singapore, pp. 341–4. Stroudsburg, PA: Association for Computational Linguistics.

Whittaker, E. W. D., and Raj, B. 2001. Quantization-based language model compression. In *Proceedings of 7th European Conference on Speech Communication and Technology (EUROSPEECH'01)*, Aalborg, Denmark, pp. 33–6. International Speech Communication Association, ISCA archive, http://www.isca-speech.org/archive/eurospeech_2001

Yuret, D. 2008. Smoothing a tera-word language model. In *Proceedings of 46th Annual Meeting of the Association for Computational Linguistics and Human Language Technology Conference (ACL-08: HLT)*, Columbus, OH, USA, pp. 141–4. Stroudsburg, PA: Association for Computational Linguistics. Software available at http://denizyuret.blogspot.com/2008/06/ smoothing-tera-word-language-model.html

Zens, R., and Ney, H. 2007. Efficient phrase-table representation for machine translation with applications to online MT and speech translation. In *Proceedings of The Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT-2007)*, Rochester, NY, USA, pp. 492–9. Stroudsburg, PA: Association for Computational Linguistics.

Appendix: List of platforms used in evaluations

A. Server: Linux kernel 2.6.32 (Centos), 64-bit, 64-GB RAM, Intel® Xeon®, 16 CPUs, 3.60 GHz, 12-MB cache, hard disk transfer 1300 MB/second.
B. Server: Linux kernel 2.6.32 (Fedora), 64-bit, 32-GB RAM, Intel® Xeon®, 8 CPUs, 2.66 GHz, 4-MB cache, hard disk transfer 400 MB/second.
C. Server: Linux kernel 2.6.27 (Fedora), 64-bit, 16-GB RAM, AMD Opteron™ 248, 2 CPUs, 2.20 MHz, 1-MB cache, hard disk transfer 200 MB/second.
D. Laptop: Windows 7™, 64-bit, 4-GB RAM, AMD Turion™ II Dual-Core M520, 2.30 GHz, 1-MB cache.
E. Desktop: Windows 7™, 32-bit,, 4-GB RAM Intel® i7®, Dual-Core, 2.93 GHz, 2-MB cache.