# Online Template Induction for Machine-Generated Emails

Michael Whittaker*
UC Berkeley
Berkeley, CA, USA
mjwhittaker@berkeley.edu

Nick Edmonds
Google
Mountain View, CA, USA
nge@google.com

Sandeep Tata
Google
Mountain View, CA, USA
tata@google.com

James B. Wendt
Google
Mountain View, CA, USA
jwendt@google.com

Marc Najork
Google
Mountain View, CA, USA
najork@google.com

## ABSTRACT

In emails, information abounds. Whether it be a bill reminder, a hotel confirmation, or a shipping notification, our emails contain useful bits of information that enable a number of applications. Most of this email traffic is machine-generated, sent from a business to a human. These business-to-consumer emails are typically instantiated from a set of email templates, and discovering these templates is a key step in enabling a variety of intelligent experiences. Existing email information extraction systems typically separate information extraction into two steps: an *offline* template discovery process (called template induction) that is periodically run on a sample of emails, and an *online* email annotation process that applies discovered templates to emails as they arrive. Since information extraction requires an email's template to be known, any delay in discovering a newly created template causes missed extractions, lowering the overall extraction coverage. In this paper, we present a novel system called Crusher that discovers templates completely online, reducing template discovery delay from a week (for the existing MapReduce-based batch system) to minutes. Furthermore, Crusher has a resource consumption footprint that is significantly smaller than the existing batch system. We also report on the surprising lesson we learned that conventional stream processing systems do *not* present a good framework on which to build Crusher. Crusher delivers an order of magnitude more throughput than a prototype built using a stream processing engine. We hope that these lessons help designers of stream processing systems accommodate a broader range of applications like online template induction in the future.

*Work done while the author was at Google.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
*Proceedings of the VLDB Endowment,* Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342264

**PVLDB Reference Format:**
Michael Whittaker, Nick Edmonds, Sandeep Tata, James B. Wendt, and Marc Najork. Online Template Induction for Machine-Generated Emails. *PVLDB*, 12(11): 1235-1248, 2019.
DOI: https://doi.org/10.14778/3342263.3342264

## 1. INTRODUCTION

People receive hundreds of billions of emails every day [34]. This data contains an abundance of useful information – online purchases, travel bookings, newsletters, updates from services such as renewal and payment reminders, appointment confirmations, and even personalized promotional offers. Mining this information enables a number of novel applications:

- The Google Assistant can answer queries such as "When is my next bill due?" using information mined from bill reminder emails [31].
- If a user walks by a store for which they have received a promotional offer email, the Google Assistant can proactively remind them of the offer using information mined from the email [31].
- Email clients can group causally related emails together into a single email thread [5]. For example, an order confirmation and corresponding shipping notification can be grouped together.

Most emails found in the wild are *not* handwritten emails sent between two humans. Rather, 90% of email traffic consists of machine-generated business-to-consumer (B2C) emails instantiated from a collection of a few million email templates, as illustrated in Figure 1. Existing email mining systems take advantage of this templatic nature of emails to extract information from emails using a two-step process [11, 20, 21, 31].

In the first step, these systems attempt to deduce the templates that businesses use to instantiate their business-to-consumer emails, a process known as **template induction**. To perform template induction, these systems cluster a sample of emails together using a carefully designed fingerprint of the email as the grouping key. This key[1] groups two

[1]Section 2.1 describes examples of these email fingerprints, including the locality-sensitive hash of the email structure that is used in Crusher.

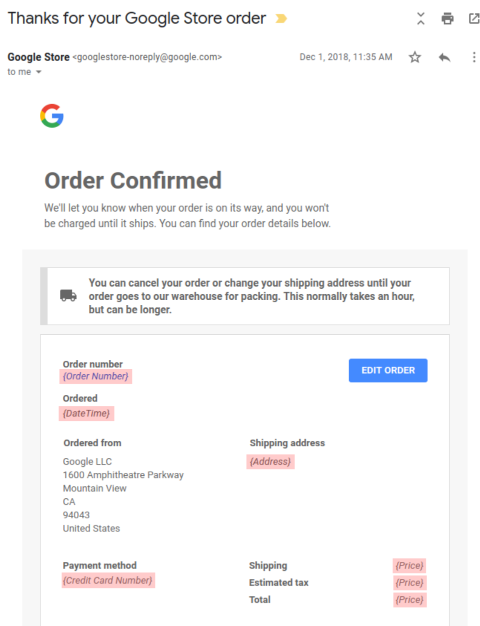



**Figure 1:** Screenshot of an order confirmation template. Personalized field locations and their types (e.g., order number, shipping address, price, etc.) are highlighted in red.

emails together if it is likely that they are instantiated from the same template. Then, the systems generate an email template for every email cluster and annotate the template with various descriptive metadata (e.g., What language is the template written in? Is the template spam? Is the template a hotel confirmation? If so, where in the template are key fields like the address, check-in date, and check-out date?).

In the second step, **email annotation**, email mining systems use the templates that were computed using template induction to annotate other emails. In particular, given an email to annotate, an email mining system computes the email's fingerprint to determine the corresponding template and then annotates the email with information from the template (e.g., the email's language, whether the email is spam, whether the email is a hotel confirmation, and if so, the check-in date). Downstream applications can then use the information stored in the email's annotations.

More formally, template induction is the problem where we are given a set of emails $E = \{e_1, e_2, e_3, ...\}$ and we want to learn a function $f$ such that $f(e_x) = f(e_y)$ iff $e_x$ and $e_y$ were instantiated from the same underlying template or script. Email annotation on the other hand is the (simpler) problem where given an email $e_x$, the annotator $g$ returns $f(e_x)$ and optionally other data associated with this template.

Traditionally, email mining systems separate these two logical steps (i.e. template induction and email annotation) into two distinct computational operations [20, 31]. Template induction is performed *offline* in batch, while email annotation is performed *online* as new emails arrive into the system. For example, template induction can be run once every week on a sample of the most recent emails, and emails can be annotated in real time as they are received.

Offline template induction has a number of inherent disadvantages. Chief among these is the matter of large template induction delays. There is a delay between when a business creates a new email template and when an email mining system discovers the template. Any emails instantiated from the template that are sent during this period of time cannot be annotated. By performing template induction offline, this delay can grow to the order of a week, leading to a number of unpleasant user experiences (e.g., the Google Assistant telling someone that they do not have any bills due, even though they received a bill reminder email a couple of days earlier).

In this paper, we present a fully online email mining system that greatly reduces this delay by performing both template induction and email annotation online. Our system, called Crusher, is a redesign of Juicer, an email mining system that was developed at Google and previously described in [31]. Whereas Juicer performs template induction offline and email annotation online, Crusher performs both online.

The benefits of online template induction in terms of discovering a template sooner are obvious. However, this advantage comes at the cost of system complexity. Crusher has to cluster, templatize, and annotate billions of emails in real time all while providing high throughput, low-latency, scalability, availability, and user privacy. In contrast, Juicer uses a single large MapReduce job for template induction.

We initially attempted to sidestep many of these complexities by implementing Crusher using an existing stream processing system such as Google MillWheel [6], Spark Streaming [38], Apache Storm [35], Apache Flink [15], or Apache Beam [7]. Unfortunately, we found that many of these stream processing systems were lacking the features—namely, complex triggers and mature stateful processing—that we needed to adequately implement Crusher. In Section 5, we share our experience and elaborate on these features. We believe that implementing these features in existing stream processing systems would enable a large class of applications.

In summary, we make the following contributions:

- We describe the task of online template induction, a novel workload core to email mining that has not previously been described in the data management literature.
- We present the design and implementation of Crusher, a planet-scale email mining system that performs template induction and email annotation online.
- We explain several non-obvious lessons that we learned through our experience of (unsuccessfully) implementing Crusher using existing stream processing systems. In particular, we provide insights into the features that, if present in existing stream processing systems, would enable a larger class of applications, including ours.
- We report that Crusher discovers 1.5 million more templates weekly, serves them to the template annotation system with a delay of minutes rather than days, and delivers a resource savings of 58% CPU time, 93% memory, and 90% disk relative to Juicer.

## 2. EMAIL CLUSTERING

In this section, we elaborate on how email mining systems like Juicer and Crusher extract information from emails using a combination of template induction and email annotation. For now, we defer discussion of whether template

induction is performed offline or online. We instead focus on the abstract algorithm behind template induction and describe how applications make use of template annotations.

## 2.1 Template Induction

Template induction consists of two parts. **Email clustering** groups emails together based on the likelihood that they were instantiated from the same template. **Template formation** processes each email in these clusters by using a set of user-defined functions to create and annotate a template.

**Email Clustering** A central idea in template induction is to generate a fingerprint for emails so that all emails generated from the same template can be easily mapped to the same fingerprint. This fingerprint can then be used as a key for the set of known templates. Several strategies have been described in the literature for how to compute such a fingerprint [5, 21]. One approach [5] is to use the fact that most instantiations of a single template are delivered from the same email address (e.g., googlestore-noreply@google.com) with subject lines that can be represented by fixed terms and wildcards (e.g., "Your order of ⟨wildcard⟩ has shipped!"). A hash of the sender and a canonicalized subject expression has been shown to be a good way to compute a template key (or fingerprint) [5].

Juicer and Crusher make the assumption that emails derived from the same original template share similar structure, as manifested by their HTML DOM trees. We convert the DOM tree of a given email into a set $S$ of XPaths through an in-order traversal. Our XPath representation only includes HTML tags and their indices, omitting all HTML attributes and values. For example, an XPath of a cell in the second row and third column of a table might be

```
/html[0]/body[0]/table[0]/tr[1]/td[2]
```

Note that since we omit all content, $S$ is representative of the *structure* of the email only. We then draw three subsets from $S$ using MinHash [14], a method for consistent sampling of sets. Each subset is represented by a hash of its elements, i.e. a numeric value. We compute three MinHash values, each with a different (but fixed) seed to get three different (but consistent) hash values. For email clusters that are exactly identical in structure, this means that three identical hash values will be obtained. For emails that have slight deviations from one another, but belong to the same template, there is a high probability that at least one of these three hashes will coincide with one another, and thus the template can be born from clustering the emails on that hash and discarding the remaining hashes. We use MinHash instead of an exact hash of the XPaths to allow for similar but not identical emails to be clustered together. For example, a purchase receipt email from an online retailer might list a variable number of items in the templated email sent to its users. By using MinHash, we prevent these slight differences from dividing the clusters.

**Template Formation** After grouping emails together based on their fingerprints, a template is formed if and only if the email cluster passes the $k$-anonymity property. In our case, the cluster of emails with the same fingerprint must contain emails sent to at least $k$ unique recipients. If the cluster satisfies this constraint, we consider it a representative cluster of a *B2C template* and its unique MinHash as its *template ID*.

The cluster of emails is then sent to a set of **user-defined cluster aggregators** (UDCAs), each of which has the opportunity to process all the emails in the cluster and return a single *template annotation* that is appended to the template and available during email annotation. Aggregating information over a cluster of emails is a powerful enabler. For example, consider date disambiguation. The date "1/2/2019" is ambiguous; it is unclear if it corresponds to the 2nd of January or the 1st of February. A date disambiguation UDCA may extract all unambiguous dates contained in a cluster's emails and record the format of those dates in the template. An email annotator, given an email with an ambiguous date, can look up the email's template and use its date format annotation to disambiguate the date. Additional UDCAs and their applications are described in the following section.

## 2.2 Applications of Templates

Template induction has proven to be useful for a variety of applications. In fact, the template ID itself is a useful signal for tasks such as spam and phishing detection, causal threading [5], and email search ranking [13].

UDCAs and the template annotations they produce have also enabled a number of applications that were previously impossible or very difficult to perform without templates, such as the previously described date disambiguation UDCA.

For example, processing the cluster of emails belonging to a single template enables the ability to determine portions of these emails that are *fixed* across all instantiations. Resultant *fixed text* and even *fixed images* have been shown to be useful for classification tasks [30, 37].

Information extraction over emails, described in detail in [31], critically relies on the UDCA infrastructure in determining the vertical, or category, of templates (e.g. purchase receipt, hotel confirmation, etc.), followed by determining for that email's template a set of extraction rules that indicate the location of variable fields pertinent to the given vertical, such as the check-in and check-out dates and address for a hotel confirmation template.

Vertical labeling is done through a UDCA that executes a set of machine-learned classifiers over each email in a template cluster that labels each email in the cluster as a *hotel confirmation*, *purchase receipt*, *bill reminder*, etc. Once all emails have been processed, the template is annotated with the label that is the most frequent across the cluster and is above a predetermined threshold.

A subsequent UDCA is applied to the cluster of emails and uses a set of field classifiers to determine the locations of the fields that are pertinent to the given vertical. For example, in the *hotel confirmation* case, we apply *check-in*, *check-out*, *hotel name*, and *hotel address* field classifiers to the XPaths of each email. Note that not all XPaths are processed; instead we utilize generic annotations, such as dates, times, and addresses to focus on promising XPaths. Similar to vertical labeling, each field classifier is applied to every XPath in every email, producing a probability. For each field, the XPath with the highest average probability of containing that field is recorded as the field's location, resulting in an extraction rule.

## 2.3 Discussion

The effectiveness of email clustering and template formation depends on the size and recency of the clusters. If the clusters are too small, then (a) we run the risk of $k$-

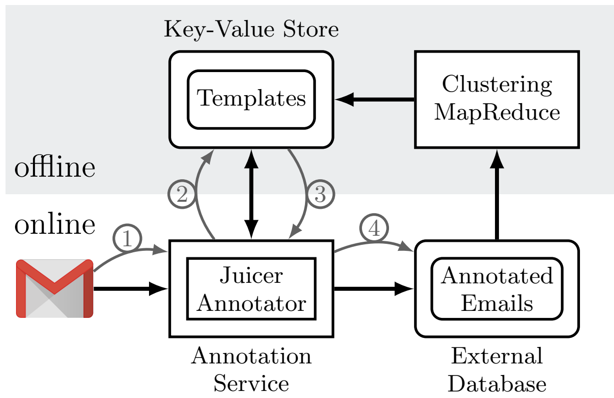



**Figure 2:** Juicer's architecture. Compute is shown in rectangular boxes, while storage is shown in boxes with rounded corners. A typical workflow is illustrated with numbered arrows.

anonymization suppressing the template and (b) relatively few emails being available to the UDCAs, resulting in less reliable vertical classification and field extraction rules. Thus, bigger clusters are better.

Moreover, if clusters are too old or too stale, then the annotations produced by the UDCAs might deteriorate over time as templates evolve. For example, consider field extraction. If a template changes the location of a field from one XPath to another, the locality-sensitive nature of the MinHash may still allow for new emails to map to that template ID, however not all of the underlying template annotations may work for the new email. For example, the date disambiguation strategy may still hold, however the field extraction rules, which rely on exact XPath matching, may no longer trigger since the field has been moved to a new location. Thus, we must re-induce templates periodically over newer emails in order to prevent staleness.

# 3. Juicer

In Section 2, we motivated *why* email mining systems perform template induction. We now turn our attention to the matter of *when* (and consequently *how*) to perform it. In this section, we summarize the design of Juicer, a system that performs template induction *offline* using a MapReduce that is run once every week. Juicer has been running in production for five years and has been previously described in [31]. We then discuss the shortcomings and limitations of this offline approach to template induction. In the next section, we present an *online* approach, implemented in Crusher, that overcomes these limitations.

## 3.1 Architecture

Juicer's architecture is illustrated in Figure 2. We describe how Juicer performs email annotation and then how it performs template induction.

**Email Annotation** When Gmail receives an email message, this message is sent to a stateless **Annotation Service**. The Annotation Service consists of a number of deterministic **annotators**. Each annotator is a piece of code that optionally attaches a piece of metadata, called an annotation, to the email. The massive scale and online nature of the Annotation Service imposes strict latency requirements. After an email has been completely processed by the set of annotators, the email (along with its annotations) is persisted in an external database. A number of downstream applications (e.g., email search, spam and phishing detection, Google assistant queries, proactive notifications, etc.) use the annotated emails persisted in this database.

One of the annotators in the Annotation Service is the **JuicerAnnotator**. When the JuicerAnnotator receives an email, it computes the email's three MinHash values, as described in Section 2. The JuicerAnnotator then issues get requests to a read-optimized key-value store with these MinHash values as keys. We'll see momentarily that after Juicer performs template induction, it stores templates in this key-value store, with each template keyed by its corresponding template ID. The key-value store is optimized for fast reads, but also supports infrequent batch updates to large ranges of keys. If the JuicerAnnotator finds a template in the key-value store for the email, then the JuicerAnnotator annotates the email with the template's vertical and uses the template's extraction rules to extract the relevant fields.

**Template Induction** The key distinguishing feature of Juicer's architecture is that it performs template induction offline. To do so, Juicer runs a weekly MapReduce (see Algorithm 1) that computes templates from 0.5% of the last 90 days of Gmail messages that are stored in the external database described above. The map phase of the MapReduce computes the MinHash values of the input emails. The reduce phase collects emails with the same MinHash value, determines if they pass the $k$-anonymity threshold (i.e. are sent to at least $k$ distinct recipients), and if so, forms templates as described in Section 2. The output of the MapReduce replaces the contents of the key-value store.

**Algorithm 1:** Juicer MapReduce pseudocode.

**Input:** A set of email documents $D, d \in D$.
**Output:** A set of Juicer templates $T, t \in T$.

```
1 Function Map(doc d):
2     hashes [h1, h2, h3] ← MINHASH(d)
3     foreach h ∈ [h1, h2, h3] do
4         EMIT(h, d)
5     end
6 Function Reduce(hash h, docs [d1, d2, ...]):
7     template t ← FORMTEMPLATE([d1, d2, ...])
8     EMIT(t)
```

## 3.2 Limitations

The offline approach to template induction has the obvious advantage of simplicity and scale. MapReduce is well-understood and designed to scale to large data parallel workloads like template induction. By creating templates outside of the latency-bounded Annotation Service, we avoid many of the availability and scalability challenges that come with developing an online template induction service. However, there are also a number of drawbacks to Juicer's offline design. These drawbacks can be roughly grouped into *latency*, *recall*, and *efficiency* limitations.

**Latency** The batch nature of Juicer's email clustering stage limits the frequency with which clustering can be executed within a given resource footprint. Adding resources can increase this frequency, but frequency scales at best linearly with resource requirements and in practice worse due to the skewed distribution of emails over templates. This

means that there is a delay between when a template is created by a sender and when Juicer forms a corresponding template that it can use to annotate emails.

This delay can negatively affect many applications that use Juicer. For example, if a bank creates a new template for their bill reminder emails, and a user asks the Google Assistant about their upcoming bills, the Assistant may not know about bill reminder emails instantiated from the recently created template. Similarly, retail stores often offer seasonal promotions that expire quickly (e.g., 50% off tortilla chips before Super Bowl Sunday, 10% off wrapping paper before Christmas). If Juicer does not detect these templates quickly enough, the promotion may expire before a user can be notified.

**Recall** Sampling is one means of reducing the latency of Juicer. In order to terminate within a week, our weekly MapReduce processes only a 0.5% sample of emails from the last 90 days. Because of this, the weekly MapReduce can sometimes fail to create a template for small clusters of emails that have been sampled below Juicer's $k$-anonymity threshold, which is adjusted conservatively to the sample rate, i.e. substantially larger than 0.5% of $k$. The offline implementation of Juicer trades some loss in recall of templates with few observed emails for lower latency in discovering more popular templates. For comparison, we ran a MapReduce job over 100% of emails from the last 90 days – while this job discovered 6 times more templates compared to the weekly job, it took nearly a month to complete. This was both prohibitively expensive and resulted in an undesirably high latency for discovering templates.

**Efficiency** Lastly, it is straightforward to observe that the MapReduce used to compute Juicer templates redundantly processes the same emails week after week until they age out of the 90 day window. The lack of state between executions of the weekly MapReduce means that the only way to refine existing templates and discover new ones, while respecting data retention limits, is to re-cluster candidate emails.

We note that some of these limitations are particular to Juicer and are not fundamental to the offline approach to template induction. For instance, we considered an incremental MapReduce approach in which the Mapper would discard emails that matched an existing template in the key-value store. While this would speed up the Shuffle and Reduce phases, this would have very little impact on the Map phase (which is a significant portion of the overall time). This approach would have led to some improvements in efficiency, but no significant improvements to latency, and no improvements to recall. Instead of pursuing designs based on an incremental MapReduce job that might address *some* of these problems, we instead chose to address all these shortcomings with a fully online system, described in the next section.

# 4. Crusher

In Section 3, we described Juicer's architecture, in which emails are clustered and templates are generated by a weekly MapReduce. In this section, we describe an alternate *online* approach in which emails are clustered and templates are generated continuously. The online approach, implemented in a system we call Crusher, overcomes many of the limitations of the offline approach. We begin by outlining the design considerations of our online approach and then describe Crusher's architecture.

## 4.1 Design Considerations

Here, we outline the design considerations that motivated Crusher's architecture.

**Scale** It is estimated that hundreds of billions of emails are sent every day [21, 31], leading to millions of emails being received by Gmail every second. Crusher should be able to handle this volume of traffic. An implicit goal is to intelligently manage Crusher's storage footprint so that template induction does not require us to maintain a copy of every email in the Gmail corpus.

**Latency of Template Induction** Crusher should form templates as quickly as possible. Ideally, as soon as an email cluster satisfies the $k$-anonymity threshold, Crusher should be able to induce a corresponding template. Note that the offline approach outlined in Section 3 has a delay on the order of a week.

**Privacy** When processing email, preserving privacy is paramount. An email cluster should only ever be processed if it satisfies the $k$-anonymity property. Moreover, Crusher must guarantee that no engineer has the ability to view any emails at any time.

**Skew** The majority of emails found in the wild are machine-generated emails sent from a business to their consumers [5, 26, 31]. The number of emails sent by each sender is highly skewed—a small number of senders account for a disproportionately large fraction of all emails. Crusher should handle this skew. In addition to implications for load-balancing and load hot-spots, the skew also presents challenges for efficiently inducing templates for relatively low-volume senders

**Fault Tolerance** Crusher is a fully online system, so machine failures are inevitable. Crusher should be able to handle these machine failures gracefully.

**Annotation Latency** The JuicerAnnotator, a key component of Juicer and Crusher, lives inside the Annotation Service and is called for each incoming email. The Annotation Service has strict latency requirements, and therefore the JuicerAnnotator has a very limited amount of time (computation or communication) available to spend on each email.

**Template Re-induction** As discussed in Section 2, the quality of a template induced from a cluster of emails depends on the recency of the emails in the cluster and its size. Crusher should allow for templates to be iteratively re-induced from clusters as they grow over time.

## 4.2 Architecture Overview

We now describe the Crusher architecture, which is illustrated in Figure 3. As with Juicer, emails are initially sent to the Annotation Service and processed by the JuicerAnnotator. Upon receiving an email, the JuicerAnnotator computes the email's three MinHash values. As with Juicer, it then attempts to fetch the template corresponding to any of these values from the external key-value store. As 90% of email traffic is machine-generated, most template lookups are successful. If the corresponding template exists, then the JuicerAnnotator uses the template to annotate the email. The Annotation Service then forwards the annotated email to an external database of annotated emails.

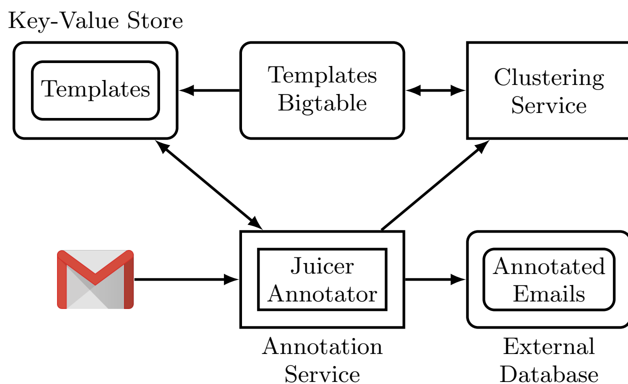



**Figure 3:** Crusher's architecture.

With Crusher (unlike with Juicer), the JuicerAnnotator also forwards the email to an online **clustering service** and then begins processing the next email immediately. It does not wait for a response from the clustering service. This "fire-and-forget" approach allows the JuicerAnnotator to meet the strict latency requirements imposed by the Annotation Service.

The clustering service is composed of a set of **clustering servers**. Clustering servers are responsible for clustering together emails with the same MinHash and for forming a template from an email cluster once it satisfies the $k$-anonymity threshold. Clustering servers persist emails and templates in an external `templates` Bigtable [17]. We first describe the schema of the `templates` Bigtable and then describe how clustering servers operate.

## 4.3 Templates Bigtable

The schema of the `templates` Bigtable is illustrated in Figure 4. Every row of the `templates` Bigtable is keyed by a MinHash and corresponds to a single email cluster and corresponding (potential) template. For example, Figure 4 shows two email clusters with MinHash `42` and `9001`.

Emails are stored in the `email` column family, and every email is assigned a unique email ID that is used as the column name. For example, Figure 4 shows three emails `"Dear Ava"`, `"To Ava"`, and `"Dear Bob"` with IDs `100`, `200`, and `300` respectively. Email `100` and `300` have MinHash `42`, and email `"To Ava"` has MinHash `9001`.

Templates are stored in a single column named `template` in the `template` column family. In Figure 4, row `42` has a template induced from emails `100` and `300`. Row `9001` does not yet have a template, which is possible if the email cluster does not yet satisfy the $k$-anonymity requirement.

In addition to the `template` and `email` column families, the `templates` Bigtable also includes a column family named `user` that is used to check whether an email cluster satisfies the $k$-anonymity threshold. This is a performance optimization that is described below. Consider an email with email ID $e$ and MinHash $h$, sent to a recipient with user ID $u$. The email is stored in row $h$ and column $e$ of column family `email`, and an empty string is stored in row $h$ and column $u$ of column family `user`. For example, in Figure 4, row `42` has entries for user `40` (Ava) and user `50` (Bob). Similarly, row `9001` has an entry for user `40`.

Note that for a given row in the `templates` Bigtable, the number of entries in the `user` column family is precisely the number of unique recipients for the email cluster. Thus, a clustering server can check whether an email cluster satisfies the $k$-anonymity property by counting the number of entries in the cluster's `user` column family. Checking the $k$-anonymity threshold is performed frequently, so we apply a number of optimizations including client-side caching of threshold check results and pinning required Bigtable columns in memory. In addition to the $k$-anonymity threshold each row also has a maximum row size threshold which applies after the $k$-anonymity threshold is met. This allows recent emails to be stored as older emails expire, ensuring a fresh set of emails for use during template re-induction, and also bounds maximum row size. Checking whether the maximum row size has been reached uses similar optimizations to the $k$-anonymity threshold check.

Also note that we apply a time-to-live (TTL) on entries in the `templates` Bigtable. If an entry of the Bigtable is older than the TTL, it is automatically deleted.[2] This TTL limits the storage footprint of the `templates` Bigtable and is a key component of template re-induction, described in more detail in the next subsection.

## 4.4 Clustering Service

When a clustering server receives an email for a particular MinHash $h$, it first checks a cached set of MinHash values called the **full template cache**. The full template cache stores a subset of the MinHash values for which a template has been induced and for which the maximum permitted number of emails have been stored in the `templates` Bigtable. That is, if MinHash $h$ is in the full template cache, then there exists a template for email cluster $h$, and the maximum permitted number of emails have been stored in row $h$. If $h$ is not in the template cache, then there may or may not exist a template in the `templates` Bigtable. The full template cache is implemented as a bounded-size LRU cache with per-element TTLs.

If a clustering server finds $h$ in the full template cache, then it simply ignores the email. Otherwise, it checks the email count for minhash $h$. If $h$ exceeds the maximum permitted email count it updates the full template cache and ignores the email, otherwise it stores the email in the `email` column family of row $h$ in the Bigtable and updates the `user` column family. It then fetches two things from the Bigtable: the existence of a template in row $h$, and the number of entries in the `user` column family of row $h$.

If a template exists in row $h$, then the clustering server checks its creation time to see if it is eligible for re-induction. Otherwise, the clustering server uses the user counts to check whether the email cluster satisfies the $k$-anonymity threshold. If the template is not eligible for re-induction or the $k$-anonymity threshold is not met, we are done—the clustering server stops processing the email. Otherwise, the clustering server fetches all of the emails in row $h$, re-checks the $k$-anonymity threshold, and invokes the UDCAs to compute a template. It then stores the template in the `template` column family of row $h$. Pseudocode for a clustering server is given in Algorithm 2. Note that over time, entries in the `templates` Bigtable and entries in the full template cache exceed their TTL and are removed. This leads to periodic template re-induction.

[2]In practice the time for which we store the email is informed by a combination of applicable regulations, business policies, and the cost of temporary storage.

| | template | email | | | user | |
|---|---|---|---|---|---|---|
| | :template | :100 | :200 | :300 | :40 | :50 |
| 42 | "Dear {name}" | "Dear Ava" | | "Dear Bob" | "" | "" |
| 9001 | | | "To Ava" | | "" | |

on disk (template, email) — in memory (user)

**Figure 4:** `templates` Bigtable schema.

**Algorithm 2:** Crusher clustering server pseudocode.

1 **Input:** The minimum number of users $k$ and maximum number of emails $S$ per template. The maximum age of a template $A$.
2 **Function** `ProcessEmail`(email $m$):
3   hashes $[h_1, h_2, h_3] \leftarrow$ MinHash($m$)
4   **foreach** $h \in [h_1, h_2, h_3]$ **do**
5     `CheckTemplateEligibility`($m$, $h$)
6   **end**
7 **Function** `CheckTemplateEligibility`($m$, $h$):
8   **if** $h$ in full template cache **then**
9     **return**
10   Update row $h$ of `templates` Bigtable with $m$
  /* $s$ is the number of emails */
  /* $t$ is whether a template exists */
  /* $n$ is the number of users */
11   Fetch $s$, $t$, $n$ from `templates` Bigtable
12   **if** $n \geq k$ **and not** $t$ **then**
    /* Induce a template. */
13     `FormTemplate`($h$)
14   **if** $n \geq k$ **and** $t$ **and** age($t$) $\geq A$ **then**
    /* Re-induce a template. */
15     `FormTemplate`($h$)
16   **if** $n \geq k$ **and** $s \geq S$ **then**
17     Update full template cache with $h$
18 **Function** `FormTemplate`($h$):
19   Fetch emails for row $h$ from `templates` Bigtable
20   Re-check $k$-anonymity thresholds and invoke UDCAs to form template from emails
21   Insert the template into `templates` Bigtable

To take advantage of the multi-core machines on which clustering servers run, we could naively spawn multiple threads, each responsible for processing a single email. However, this naive approach to parallelizing a clustering server is suboptimal. To see why, we consider the operation of a clustering server as a composition of two main steps: the invocation of `CheckTemplateEligibility` and the invocation of `FormTemplate`. `CheckTemplateEligibility` persists an email in the Bigtable and fetches data from the Bigtable, whereas `FormTemplate` computes a template from an email cluster. Storing and fetching data from the Bigtable is an I/O-bound operation, while executing UDCAs and computing templates is a memory- and CPU-bound operation. We use a SEDA architecture [36] to extract more parallelism from a clustering server by executing these two steps on two separate thread pools and by tuning the number of threads in each thread pool separately. Both thread pools perform admission control to avoid exhausting a server's CPU, memory, or network resources. Thus, an overloaded clustering server may selectively drop emails.

Periodically, the templates stored in the Bigtable are migrated to the key-value store that is read by the JuicerAnnotator. In theory, we could eliminate this extra template migration step, and the JuicerAnnotator could use the Bigtable directly. This would make a new template available in the Annotation Service as soon as it meets the $k$-anonymity threshold. However, the Annotation Service currently *requires* the use of the key-value store in order to support debugging and development tools for other annotators. Apart from the delays introduced by the constraints imposed by the Annotation Service, emails can be annotated with a template ID as soon as the corresponding cluster satisfies the $k$-anonymity threshold. This is usually on the order of a few minutes compared to delays of a week or longer that is present in the offline design.

## 4.5 Discussion

Our Crusher design overcomes many of the limitations of our existing Juicer design and satisfies many of the requirements outlined in Section 4.1.

**Scale** Clustering servers do not coordinate with one another, so Crusher can scale to handle a large volume of emails by increasing the number of clustering servers. The only point of contention in Crusher is the `templates` Bigtable, but fortunately Bigtable can scale to petabytes of data and thousands of machines [17].

**Latency of Template Induction** Crusher forms a template from a cluster of emails as soon as the cluster satisfies the $k$-anonymity threshold.

**Privacy** By construction, a clustering server will only ever form a template from an email cluster if the email cluster satisfies the $k$-anonymity threshold. The TTL on the `templates` Bigtable also ensures that emails are not persisted indefinitely. Data are encrypted, and can only be accessed by a role account that is limited to running the Crusher binary built from code that is reviewed, submitted, and auditable. Nobody involved with the project had access to visually inspect any of the data.

**Skew** As noted earlier, a small number of businesses send a disproportionately large fraction of emails. As a result, a small number of templates account for a relatively large fraction of emails. Crusher handles this skew by caching frequently accessed template metadata in the clustering servers to reduce Bigtable reads.

**Fault Tolerance** Crusher inherits the fault tolerance of the systems it is built on (e.g., the Annotation Service, Bigtable, the External Database). The only Crusher component that does not inherit fault tolerance is the clustering service. Fortunately, Crusher does not need to implement any additional mechanisms to ensure fault tolerance for the clustering service because it is naturally fault tolerant. Online email clustering is naturally tolerant to email loss because an email template can be formed from *any* representative sample of emails instantiated from the template. The

quality of the template increases with the number of emails, but forming a template from *every* email is prohibitively expensive and largely unnecessary due to diminishing returns. Thus, failed clustering servers can simply be restarted without the need to recover any buffered emails. A clustering server failure simply changes the sample of emails used to instantiate a template. Note that Juicer similarly does not process *every* email, only a small sample.

**Annotation Latency** The JuicerAnnotator sends fire-and-forget requests to the clustering service, so that the latency of the JuicerAnnotator is completely independent of the latency of the clustering service. This helps keep the latency of the JuicerAnnotator in the Annotation Service low.

**Template Re-induction** Crusher uses TTLs on the `templates` Bigtable and on the full templates cache to periodically re-induce templates.

# 5. STREAM PROCESSING JUICER

Before we designed and implemented Crusher, we considered a different (and arguably more natural) approach to converting Juicer's batch MapReduce job to an online job: stream processing systems.

Stream processing systems [6, 15, 24, 28, 35, 38] generalize data-parallel batch processing jobs that operate over fixed, finite data sets to data-parallel stream processing jobs that operate over streaming, potentially infinite data streams. At a glance, these systems appear to be the perfect tool to overcome the limitations of Juicer's offline approach to template induction. All we'd have to do is rewrite our weekly MapReduce job using a stream processing system, right?

Unfortunately not. We developed several prototype implementations that focused on replacing the weekly MapReduce job with a stream processing job. Through our experience, we found that many existing stream processing systems lack the features required to satisfy all of our system requirements. In particular, we found that many stream processing systems lack the complex windowing and triggering semantics required to implement Crusher efficiently. Fortunately, many stream processing systems provide an API to streaming operators that allows them access to persistent state across invocations. This API makes it possible to simulate complex windowing and triggering semantics, but we found that while it's possible, it is often inefficient. We hope that this section motivates future research on stream processing systems that will enable a broader range of real-world streaming applications like Crusher.

## 5.1 Windows and Triggers

Stream processing systems allow developers to discretize infinite data streams into finite groups of data called *windows*. Stream processing systems typically support some subset of the following four types of windows:

- *Global windows.* With global windowing, every stream element is assigned to a single global window.
- *Fixed windows.* Fixed windows are disjoint equally-sized windows in which one window begins the moment the previous window ends (e.g., hourly windows that begin every hour).
- *Sliding windows.* Sliding windows are equally-sized windows that are not necessarily disjoint. One sliding window begins after some fixed delay after the previous window begins (e.g., hourly windows every thirty minutes).
- *Session windows.* With session windows, stream elements are grouped in a given window such that no two consecutive elements are separated by more than a fixed duration (e.g., five minutes).

Many stream processing systems also allow developers to *trigger* the materialization of a growing window, allowing it to be processed by downstream operators more often than just at the end of the window. The triggers supported by existing stream processing systems are typically based on count or time, either triggering a window after the first $n$ entries have been received, after every $n$ entries have been received, or after some time after the end of the window. The data associated with a window is not garbage collected until after the window has been triggered for the last time.

We found that existing windows and triggers are insufficiently expressive to implement the clustering service naturally and efficiently. Ideally, we could group emails together by their MinHash, window them into email clusters, and trigger a window as soon as its email cluster satisfies the $k$-anonymity threshold. Unfortunately, while most stream processing systems support triggers based on count (e.g., trigger a window after $k$ emails), we did not find any existing stream processing systems that include built-in triggers based on *distinct* counts (e.g., trigger a window after $k$ unique users).

This makes it challenging to implement the clustering service using windows and triggers. For example, consider the pseudocode implementation of the clustering service shown in Figure 5. The implementation begins with a stream of emails called `emails`. It then uses the `ExtractMinHashes` function to map every email $m$ to the pairs $(h_1, m)$, $(h_2, m)$, $(h_3, m)$ where $h_1, h_2, h_3$ are the MinHash values of $m$. It then windows this stream into fixed $n$ day windows for some constant $n$, triggering the window (with accumulation) every $k$ emails. Finally, it groups emails by their MinHash and forms a template from an email cluster if it satisfies the $k$-anonymity threshold.

This implementation has a number of serious limitations. First, the window is triggered only every $k$ emails and at the end of every $n$ days. Imagine that $k = 1,000$ and that $1,000$ emails arrive for a particular email cluster with only 999 unique users. The trigger is fired, but the emails do not satisfy the $k$-anonymity threshold because there are only 999 unique users. Then, imagine the $1,001^{\text{st}}$ email arrives with the $1,000^{\text{th}}$ unique user. At this point, the $k$-anonymity threshold is met, so a template could be formed, but the trigger will not fire again until 999 more emails arrive or until $n$ days have elapsed. Decreasing $n$ decreases the upper bound on this template induction delay, but if $n$ is too small, then some slowly accumulating email clusters will be prematurely garbage-collected before the clustering service has the opportunity to form a template.

Alternatively, instead of triggering the $n$-day windows every $k$ emails, we could trigger them every $m$ emails for some small value of $m$ (e.g., 1 or 2). With this approach, there is less delay between when a cluster satisfies the $k$-anonymity threshold and when a template is formed from the cluster. However, this approach triggers the window unnecessarily often, leading to the email cluster being transferred through

```
emails
  .FlatMap(ExtractMinHashes)
  .FixedWindows(Days(n)),
  .TriggerWithAccumulation(Repeat(Count(k)))
  .GroupByKey()
  .Filter(CheckKAnonymity)
  .Map(FormTemplate)
```

**Figure 5:** Pseudocode implementations of the clustering service using fixed windows.

the stream processing job significantly more than is necessary. For example, if $k = 1,000$ and $m = 1$, then the email cluster is triggered three orders of magnitude more often. Moreover, even after an email cluster satisfies the $k$-anonymity threshold, it will continue to trigger after every $m$ emails enter the cluster.

As another alternative, we can eschew the $n$-day fixed windows for $n$-hour session windows. Now, if the $1,001^{\text{st}}$ email is sent to the $1,000^{\text{th}}$ unique user, the clustering service will only have to wait up to $n$ hours to form a template. However, this implementation has its own limitations. For example, if 500 emails arrive at one point in time and 500 more arrive $n$ hours and 1 minute later, this implementation will not attempt to form a template from these $1,000$ emails even though they may satisfy the $k$-anonymity threshold.

While these are just a few possible implementations of the clustering service using a stream processing system, they are emblematic of the overall inexpressiveness of existing windowing and triggering semantics. Also note that some stream processing systems, like Apache Flink [15], allow developers to write custom triggers, but the custom trigger APIs are experimental or unstable.

This seems to suggest that stream processing systems should support more sophisticated forms of windowing and triggering. This would enable applications like Crusher. However, there is a fundamental tension between expressiveness and complexity. More sophisticated windowing and triggering semantics are both more challenging for a user to understand and more difficult for a system developer to implement. Our conversations with engineers working on stream processing engines within Google suggest that finding the right balance between expressiveness and complexity is still an ongoing area of investigation.

## 5.2 Stateful Processing

Because complex windows and triggers are largely unsupported, developers must instead turn to alternative APIs. In particular, *stateful processing* can act as a substitute for complex windows and triggers. With stateful processing, streaming operators are given access to per-key persistent state that they can read and update whenever they process a stream element.

We can use persistent state to implement a Crusher variant as follows. First, we group emails together by MinHash and stream them into a stateful map operator. The stateful map operator maintains two pieces of state: (1) the set of emails it has received thus far, dubbed `emails` and (2) the corresponding set of unique email recipients, dubbed `users`. When the stateful map operator receives an email $m$ with recipient $r$, it adds $m$ to `emails` and $r$ to `users`. It then checks to see if `users` is large enough to satisfy the privacy threshold. If it is, then the stateful map operator forms a template from the set of emails.

In our experience, we found that many (but not all) existing stream processing systems lack the features that enable this design to be implemented efficiently. Some stream processing systems, like Spark Streaming and Apache Storm, treat per-key state as an opaque blob, only allowing the state to be persisted by writing the blob in its entirety. This approach to state management works well for stream processing jobs with a small amount of state, but it performs poorly for jobs with large state. For example, imagine that we implemented our Crusher design using this approach to state management. In order to persist `emails`, our stateful map operator would read and re-write `emails` in its entirety, even if only a handful of new emails have been inserted since the last time `emails` was persisted.

Other stream processing systems like Apache Flink and Apache Beam [7] support richer state APIs that allow stateful operators to manipulate state at finer granularities. For example, Flink and Beam allow state to be modeled as a set, a bag, a list, etc. Using these state APIs, we can model `emails` and `users` as sets and can efficiently add elements to these sets without having to re-write them in their entirety.

However, even these richer APIs can be too restrictive, not allowing for some performance optimizations. For example, recall the `templates` Bigtable, presented in Section 4. The `email` and `template` column families were only partially cached in memory because they were large and accessed infrequently. The `user` column family, on the other hand, was cached entirely in memory because it was small and accessed frequently. Storm, Flink, Samza, and Beam allow state to be modeled at a finer granularity than a blob, but they don't support features that enable a developer to perform this sort of lower-level performance tuning. Note that these stream processing systems do automatically perform *some* storage optimizations (e.g., caching small objects), but they do not provide application developers the ability to perform fine-grained optimizations for performance sensitive applications.

Note that some stream processing systems like Apache Storm and Apache Flink allow a developer to implement custom state backends. This functionality allows a developer to customize the way that state is persisted, tailoring it to meet the needs of a particular application. Though this functionality makes it *possible* to implement persistent state efficiently, it doesn't make it *convenient*. Implementing and debugging a custom state backend is onerous and detracts from the convenience of using a stream processing system.

Because existing windowing and triggering functionality is insufficient to implement a Crusher variant, alternative APIs, like efficient state management, are required but largely lacking from existing stream processing systems. We argue that many applications would benefit from improved support for state management.

# 6. EVALUATION

In this section, we evaluate Crusher on production data and on a synthetic workload. We also compare Crusher against a variant of Juicer written using a stream processing system. In particular, we answer the following questions:

**Crusher in the Real World.** How does the resource usage of Crusher compare to the resource usage of Juicer? Do we discover more templates with Crusher without having to spend proportionately more resources? Do the templates

discovered by Crusher result in more emails being annotated?

**Crusher Performance Capabilities.** How well does Crusher scale with the number of clustering servers? How well does it handle data skew? How much does a clustering server's SEDA architecture improve performance?

**Crusher vs Stream Processing Juicer.** How does the performance of Crusher compare to Juicer variants implemented using a stream processing system?

## 6.1 Crusher in the Real World

Crusher is currently run in production alongside the existing Juicer system as we transition away from offline template induction entirely. Emails are sent to Crusher for clustering and potential template induction (if the $k$-anonymity threshold is met). In this fashion, we initialize Crusher with a known set of templates without the need to run Crusher for 90 days before the systems converge. At the time of writing, Crusher is in production running on 10% of traffic.

Table 1 compares the resource utilization and output of Juicer and Crusher during a one week period. Juicer's MapReduce discovered 440,933 templates in addition to those discovered by the previous week's MapReduce. During this same week, Crusher discovered 2,085,296 templates which were added to the key-value store as they were discovered and can be used to annotate emails within minutes of template formation. Crusher discovered far more new templates than Juicer because Juicer is hampered by the fact that its $k$-anonymity threshold is adjusted quite conservatively to the 0.5% sampling rate to cope with the perils of small sample sizes. Since the number of recipients of a given template follows a Zipfian distribution, there are many templates that clear the $k$-anonymity threshold but do not clear the adjusted value of $k$ employed by Juicer. Figure 6 shows that the additional templates discovered by Crusher resulted in approximately 10% more emails being annotated over the course of a recent week.

Juicer's MapReduce used resources roughly equivalent to $5,000$ CPUs with 19GB of memory each running for a day. For Crusher, in addition to the resources used by the clustering servers, we also show the resources required to serve the underlying `templates` Bigtable. The usage presented is the average over the week and is an overestimate due to the difficulty of separating some other ancillary Bigtable traffic. Crusher demonstrates a reduction of 58% CPU time, 93% memory, and 90% disk relative to Juicer while discovering and annotating more templates with lower latency.

## 6.2 Crusher Performance Capabilities

**Synthetic Workload** In this subsection, we evaluate Crusher's clustering service using a synthetic workload. The synthetic workload employs 50 client machines, each running 40 client threads. Initially, every client thread sends 10 emails per second to the clustering service. Over the course of 13 minutes, every client thread increases its sending rate by 0.15 emails per second per second. Thus, in aggregate, the clients generate 20,000 emails per second at the beginning of the workload and 254,000 emails per second at the end of the workload.

Clients send randomly generated synthetic emails. When a client generates an email, it first samples an integer from a Zipfian distribution with (tunable) parameter $\alpha$. This integer acts as the email's template ID and is embedded in

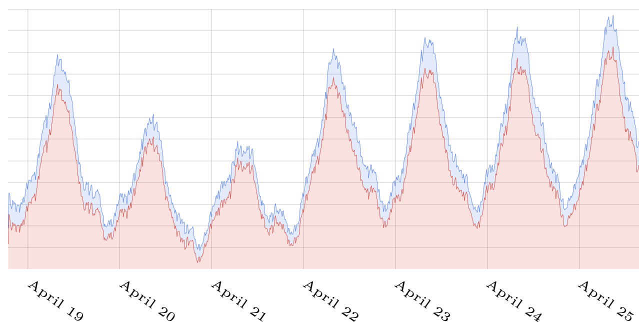



**Figure 6:** Incremental number of annotated emails by Crusher vs. Juicer. The $y$-axis is linear, and units have been removed to avoid disclosing Gmail traffic data. The red shading represents emails annotated by Juicer *and* Crusher while the blue shading represents Crusher alone. The magnitude of the annotation increase averages 10%

the email. When a clustering server receives an email with an embedded template ID, it bypasses its normal MinHash computation and instead uses the embedded ID directly. Drawing template IDs from a Zipfian distribution simulates the skewed nature of real-world emails. Next, a client randomly generates a piece of HTML for the body of the email. The size of the email is drawn from a Gaussian distribution with a mean of 10KB and a standard deviation of 2KB. If the email is larger than 30 KB, it is trimmed to 30KB.

Recall that the clustering service processes at most $S$ emails per template over the email TTL interval. We approximate this behavior by imposing the restriction that a client sends at most $T$ emails for a given template ID, for a tunable parameter $T$.

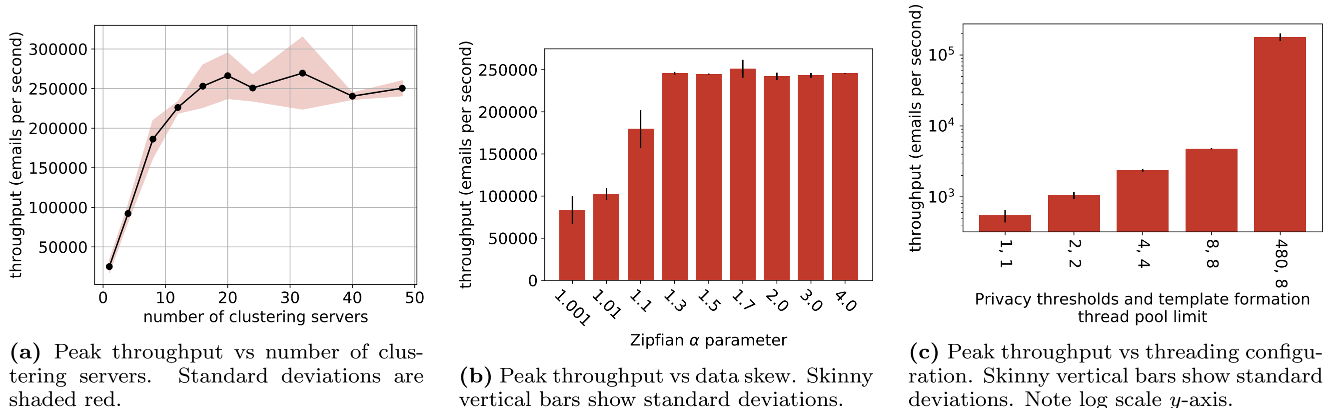
**Scalability** We evaluate the scalability of the clustering service by measuring its peak throughput against our synthetic workload when deployed with various numbers of servers. For this experiment, we set $T = 2,000,000$ and set $\alpha = 1.1$. Moreover, the two thread pools employed by the clustering servers (i.e. the one that checks the $k$-anonymity threshold and the one that forms templates, see Section 4.4) are limited to 480 and 8 threads. The $k$-anonymity threshold is set to $k = 1,000$. The results of this benchmark are shown in Figure 7a.

A single clustering server achieves a throughput of roughly $25,000$ emails per second. This throughput increases linearly with the number of servers for up to roughly 8 servers. It then increases sublinearly and approaches an upper bound of roughly $250,000$ emails per second. This is the peak client load generated by our synthetic benchmark, so the clustering service cannot exceed this upper bound. At this peak throughput, the system can process 21.6 billion emails per day. In production, 80% to 90% of emails have a known template and most of these will be served from directly the key-value store and not be processed by the clustering service. This allows Crusher to support 100 to 200 billion incoming emails a day. We expect to be able to scale further by provisioning more resources for the `templates` Bigtable.

**Tolerance to Skew** Next, we evaluate the clustering service's ability to tolerate the skewed nature of emails. To do so, we run an eight-machine clustering service deployment against our synthetic benchmark and vary the Zipfian parameter $\alpha$. All other parameters are unchanged from the previous experiment. The results are shown in Figure 7b.

**Table 1:** Weekly resource utilization of a typical Juicer MapReduce and Crusher.

| | CPU-hours | memory (GB-hours) | disk (TB) | # of templates | template delay |
|---|---|---|---|---|---|
| **Juicer** | 118,405 | 2,244,897 | 332.19 | 440,933 | 1 week |
| **Crusher** | | | | 2,085,296 | minutes |
| Clustering Service | 11,529 | 87,149 | 0.00 | | |
| Templates Bigtable (average) | 38,628 | 67,719 | 33.06 | | |




**(a)** Peak throughput vs number of clustering servers. Standard deviations are shaded red.

**(b)** Peak throughput vs data skew. Skinny vertical bars show standard deviations.

**(c)** Peak throughput vs threading configuration. Skinny vertical bars show standard deviations. Note log scale $y$-axis.

**Figure 7:** Crusher evaluation.

As the Zipfian parameter $\alpha$ increases, the skew of the email distribution increases and a larger number of emails are generated for a smaller number of templates. Fortunately, data skew does not hurt the throughput of the clustering; in fact, it improves it! The clustering service achieves a peak throughput of roughly $83,000$ emails per second for $\alpha = 1.001$, roughly $102,000$ for $\alpha = 1.01$, and roughly $179,000$ for $\alpha = 1.1$. For all larger values of $\alpha$, the peak throughput is approximately $250,000$ emails per second (the maximum client load).

Data skew positively correlates with peak throughput due to template caching. Once a clustering server caches a template, it can process subsequent emails matching this template directly from cache. The clustering server does not have to contact the `templates` Bigtable at all. Thus, when the data skew increases, the fraction of emails that are served from cache increases, and the overall throughput of the clustering service increases.

**SEDA Architecture** Recall that every clustering server employs two thread pools. The first thread pool is responsible for computing MinHash values, for storing emails in the `templates` Bigtable, and for checking whether an email cluster satisfies the $k$-anonymity threshold. The second thread pool is responsible for forming templates from email clusters that do satisfy the $k$-anonymity threshold. The first thread pool is mostly I/O-bound, while the second thread pool is both CPU- and memory-intensive.

We now evaluate the effect of separating these two responsibilities across two thread pools using a SEDA architecture. We run an eight-machine clustering service deployment against our synthetic workload with $\alpha = 1.1$ and all other parameters unchanged from the previous two experiments. We vary the number of threads in the first and second thread pool and measure the peak throughput. The results are shown in Figure 7c. Note that throughput is shown on a log scale.

Without dividing work between two thread pools, a clustering server would be forced to check the $k$-anonymity threshold and form templates with a fixed number of threads. Moreover, this fixed number of threads cannot be too large or else the number of threads concurrently forming templates becomes large enough to exhaust a machine's memory capacity. To simulate clustering servers that use a small and fixed number of threads, we limit both thread pools to 1, 2, 4, and 8 threads. The clustering service achieves throughputs of 543, 1046, 2385, and 4765 emails per second respectively. Relaxing the restriction of a small fixed number of threads, we achieve roughly 180,000 emails per second with 480 threads in the first thread pool and 8 threads in the second. This is a two order of magnitude increase in throughput.

## 6.3 Crusher vs Stream Processing Juicer

While performance benchmarking of different streaming systems on template induction is an explicit non-goal of this paper, we briefly present performance results for the simple implementation in Figure 5. The prototype implementation was built using Apache Beam-like APIs leveraging the MillWheel [6] runtime. Importantly from a performance standpoint, the MillWheel runtime is designed to provide fault-tolerant, exactly-once semantics, while Crusher explicitly tolerates failures in both email storage and template induction in order to maximize throughput. Our goal in this section is to demonstrate that Crusher is clearly competitive with an alternative implementation based on a mature stream processing framework.

Figure 8 reuses the synthetic load generator from Figure 7, writes results to Bigtable as in previous performance evaluations, and scales the number of MillWheel servers. Given the simplicity of our implementation, it is no surprise that the peak throughput is significantly lower for our streaming implementation than for Crusher. Experts in stream processing systems would likely be able to optimize our implementation to significantly improve throughput but closing the nearly 30× performance gap entirely would likely require a significant rewrite of the application and new features in the underlying engine.

The superior throughput of Crusher relative to our streaming implementation is largely attributable to three reasons:

- Crusher optimizes the common task of counting unique users using client-side caching and memory-mapped Bigtable columns. Implementing a similar optimization in the streaming prototype is not straightforward without changes to the underlying engine.

- The MillWheel query plan maps each execution stage to a separate Bigtable, increasing the number of disk writes required and reducing locality. This is a consequence of the runtime needing to support exactly-once semantics. Crusher reuses a single Bigtable.

- Crusher's SEDA architecture provides two orders of magnitude increase in throughput, cf. Figure 7c. While we allowed each MillWheel server to auto-scale its resource consumption without limit, the internal threading configuration was not exposed to our application.

In retrospect, the performance gap between our implementation on a streaming engine and Crusher is not surprising. Our observations are consistent with arguments made previously in [32] that for applications that are sufficiently different from the general stream processing setting (dashboarding and alerting over moving windows) a specialized implementation might offer a large performance advantage.

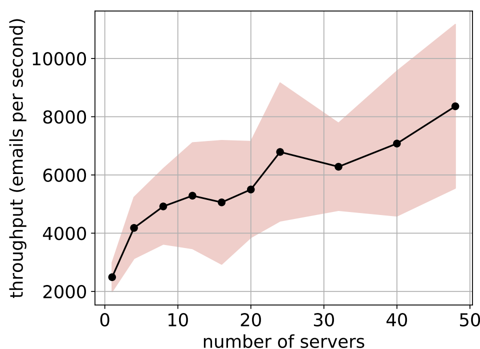



**Figure 8:** Peak throughput vs number of MillWheel servers using the simple streaming implementation from Figure 5. Standard deviations are shaded red.

# 7. RELATED WORK

**Template Induction** Template induction is the technique of identifying a skeleton of repeated content based on previously seen examples that are similar to each other. It has been widely used in information extraction over structured web pages [10, 25]. For emails, multiple algorithms for template induction have been described [5, 11, 21] along with applications in information extraction [3, 20, 31], email threading [5], and hierarchical classification [37]. This paper uses techniques previously described in the literature [31], but is the first description of an online template induction system.

**Stream Processing Systems** The data management literature contains nearly two decades of work on stream processing systems. Early systems like Aurora [2, 12], Borealis [1], STREAM [8], and TelegraphCQ [16] laid the groundwork for complex query processing over large data streams. Subsequent works have studied query processing abstractions [9] and also adapted the ideas to MapReduce-style processing [18]. Several systems have been built in industry including Google MillWheel [6], Spark Streaming [38], Oracle Stream Analytics [29] (which compiles to Spark Streaming), Apache Storm [35], Apache Flink [15], Apache Heron [24], Apache Samza [28], Naiad [27], and Spade [23]. Each system strikes various trade-offs between fault-tolerance, programming paradigm, and scalability [33]. In contrast to these systems, our system is *not* a general-purpose streaming engine, but rather a system designed for the specific application of processing a large stream of emails to detect templates. We argue that the balance between expressivity and performance provided by existing streaming systems, rather surprisingly, fails to serve this application well.

**Data Mining on Streams** While the data mining community has studied several algorithms [19, 22] for mining high-volume data streams, the focus has been on time-series data and algorithms for learning sketches. The majority of the work on mining text streams [4] has focused on detecting evolving topics on sources like the Twitter postings stream. Much of this body of work focuses on algorithms and applications rather than a general purpose system for mining. To our knowledge, our paper presents the first real-world study of a streaming system over email for mining templates.

# 8. CONCLUSION

This paper presents the problem of online email template induction. While template induction has been studied previously in the literature, to our knowledge, this is the first description of a system for template induction in an online setting. We presented the design and implementation of a system that performs online template induction on a planet-scale email service. Experiments on a synthetic workload show that the system can handle an average throughput of at least 250,000 emails per second with about 20 servers. The system is in production at 10% of traffic as of the time of writing this paper. Our experiments show that in addition to decreasing the latency of template discovery from days to minutes, Crusher discovers more templates while delivering resource savings of 53% CPU time and 93% in memory footprint compared to the batch version implemented as a MapReduce job. We also discussed reasons why we were unable to implement Crusher using an existing stream processing system. A simple prototype implementation of Juicer using an existing stream processing system with the same resource footprint produced between 10 to 40 times lower throughput. We hope that our experience inspires stream processing system developers to extend APIs to support novel applications like online template induction.


# 9. ACKNOWLEDGMENTS

This paper benefited from thoughtful inputs from engineers on stream processing infrastructure teams at Google including Dan Sotolongo, Tyler Akidau, Daniel Mills, Harsh Vardhan, and Eugene Kirpichov. We are also grateful to engineers on the Juicer team who contributed to analyses and design discussions in the early stages of the project: Jing Xie, Nguyen Vo, Qi Zhao, and Ying Sheng. Finally, we thank Jinan Lou for the early advocacy of online template induction.

## 10. REFERENCES


[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *2nd Biennial Conference on Innovative Data Systems Research*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[3] M. K. Agarwal and J. Singh. Template trees: Extracting actionable information from machine generated emails. In *International Conference on Database and Expert Systems Applications*, pages 3–18, 2018.

[4] C. C. Aggarwal. Mining text and social streams: A review. *SIGKDD Explor. Newsl.*, 15(2):9–19, June 2014.

[5] N. Ailon, Z. S. Karnin, E. Liberty, and Y. Maarek. Threading machine generated email. In *6th ACM International Conference on Web Search and Data Mining*, pages 405–414, 2013.

[6] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at Internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[7] Apache. Apache beam: An advanced unified programming model. `https://beam.apache.org/`, 2019.

[8] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The Stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[9] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[10] A. Arasu and H. Garcia-Molina. Extracting structured data from web pages. In *2003 ACM SIGMOD International Conference on Management of Data*, pages 337–348, 2003.

[11] N. Avigdor-Elgrabli, M. Cwalinski, D. Di Castro, I. Gamzu, I. Grabovitch-Zuyev, L. Lewin-Eytan, and Y. Maarek. Structural clustering of machine-generated mail. In *25th ACM International Conference on Information and Knowledge Management*, pages 217–226, 2016.

[12] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *The VLDB Journal*, 13(4):370–383, Dec. 2004.

[13] M. Bendersky, X. Wang, D. Metzler, and M. Najork. Learning from user interactions in personal search via attribute parameterization. In *10th ACM International Conference on Web Search and Data Mining*, pages 791–799, 2017.

[14] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157–1166, 1997.

[15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4):28–38, Dec 2015.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *1st Biennial Conference on Innovative Data Systems Research*, 2003.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[18] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *7th USENIX Symposium on Networked Systems Design and Implementation*, pages 313–328, 2010.

[19] J. de Andrade Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. P. L. F. de Carvalho, and J. Gama. Data stream clustering: A survey. *ACM Comput. Surv.*, 46(1):13:1–13:31, July 2013.

[20] D. Di Castro, I. Gamzu, I. Grabovitch-Zuyev, L. Lewin-Eytan, A. Pundir, N. R. Sahoo, and M. Viderman. Automated extractions for machine generated mail. In *Companion Proceedings of The Web Conference*, pages 655–662, 2018.

[21] D. Di Castro, L. Lewin-Eytan, Y. Maarek, R. Wolff, and E. Zohar. Enforcing k-anonymity in web mail auditing. In *9th ACM International Conference on Web Search and Data Mining*, pages 327–336, 2016.

[22] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy. Mining data streams: A review. *ACM SIGMOD Record*, 34(2):18–26, June 2005.

[23] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *2008 ACM SIGMOD International Conference on Management of Data*, pages 1123–1134, 2008.

[24] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250, 2015.

[25] N. Kushmerick, D. S. Weld, and R. Doorenbos. Wrapper induction for information extraction. In *15th International Joint Conference on Artificial Intelligence*, pages 729–737, 1997.

[26] Y. Maarek. Is mail the next frontier in search and data mining? In *9th ACM International Conference on Web Search and Data Mining*, pages 203–203, 2016.

[27] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *24th ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[28] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza:

stateful scalable stream processing at LinkedIn. *PVLDB*, 10(12):1634–1645, 2017.
[29] Oracle. Oracle stream analytics. https://www.oracle.com/middleware/technologies/complex-event-processing.html, 2019.
[30] N. Potti, J. B. Wendt, Q. Zhao, S. Tata, and M. Najork. Hidden in plain sight: Classifying emails using embedded image contents. In *2018 World Wide Web Conference*, pages 1865–1874, 2018.
[31] Y. Sheng, S. Tata, J. B. Wendt, J. Xie, Q. Zhao, and M. Najork. Anatomy of a privacy-safe large-scale information extraction system over email. In *24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 734–743, 2018.
[32] M. Stonebraker and U. Çetintemel. "One size fits all": an idea whose time has come and gone. In *21st International Conference on Data Engineering*, pages 2–11. IEEE, 2005.
[33] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, Dec. 2005.
[34] The Radicati Group. Email statistics report, 2018-2022. https://www.radicati.com/wp/wp-content/uploads/2018/01/Email_Statistics_Report,_2018-2022_Executive_Summary.pdf, 2018.
[35] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *2014 ACM SIGMOD International Conference on Management of Data*, pages 147–156, 2014.
[36] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *18th ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.
[37] J. B. Wendt, M. Bendersky, L. Garcia-Pueyo, V. Josifovski, B. Miklos, I. Krka, A. Saikia, J. Yang, M.-A. Cartright, and S. Ravi. Hierarchical label propagation and discovery for machine generated email. In *9th ACM International Conference on Web Search and Data Mining*, pages 317–326, 2016.
[38] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *24th ACM Symposium on Operating Systems Principles*, pages 423–438, 2013.