Conditional Cuckoo Filters

Daniel Ting
Tableau Software
Seattle, Washington
dting@tableau.com

Rick Cole
Tableau Software
Palo Alto, California
ricole@tableau.com

ABSTRACT
Bloom filters, cuckoo filters, and other approximate set membership sketches have a wide range of applications. Oftentimes, expensive operations can be skipped if an item is not in a data set. These filters provide an inexpensive, memory efficient way to test if an item is in a set and avoid unnecessary operations. Existing sketches only allow membership testing for a single set. However, in some applications such as join processing, the relevant set is not fixxed and is determined by a set of predicates.

We propose the Conditional Cuckoo Filter, a simple modification of the cuckoo filter that allows for set membership testing given predicates on a pre-computed sketch. This filter also introduces a novel chaining technique that enables cuckoo filters to handle insertion of duplicate keys. We evaluate our methods on a join processing application and show that they significantly reduce the number of tuples that a join must process.

KEYWORDS
Bloom filters, cuckoo filters, approximate set membership, semi-join

1 INTRODUCTION
Approximate set membership data sketches, such as Bloom and cuckoo filters, allow users to query if an item belongs to a given set \( S \), i.e. if \( x \in S \). If \( x \notin S \), the item is always correctly classified. However, an item \( x \notin S \) has some small probability of being incorrectly classified as being in the set. In other words, the filters return no false negatives and have a small probability of returning false positives. These sketches are useful as they provide an inexpensive way to check whether an expensive operation needs to be performed. For example, in LSM-trees [41], a disk access can be skipped if a sketch guarantees a queried key is not in an L-leaf. They are also used in join processing [8, 10, 36] and cold storage structures [1] in databases and have many applications in networking [10].

We consider the general problem of testing set membership given predicates, in particular, equality predicates with an extension to range predicates. Consider a dataset \( D \) with each row consisting of a search key \( k \) and vector of attributes \( \mathbf{a} \) where the key may not be unique. Given a predicate \( P \), we wish to test if an item \( x \) is in the set \( S_P \) of keys with attributes satisfying that predicate

\[
S_P = \{ k : (k, a) \in D \text{ and } P(a) = \text{true} \}.
\]

We show that this can have significant benefits in join processing by enabling predicates from one table to be pushed down to scans on other tables, thus reducing the number of tuples in intermediate results. Furthermore, unlike existing methods using approximate set membership filters, this can be done using pre-built filters that do not require additional table scans to construct filters at query time or communicating filters over a network. We evaluate this reduction on the real world IMDB data set and show that the number of tuples that pass the filter is less than 1/3 the original number. Other possible applications of the sketch include problems in information retrieval and access control discussed in Section 9.

We propose the Conditional Cuckoo Filter (CCF) to address this problem. The CCF can be seen as a simple modification of a cuckoo hash table [49] where, rather than storing a key, value pair, it stores fingerprints or sketches of both. It is thus similar to cuckoo filters [20] which store only key fingerprints. In a CCF, the value is a sketch of attribute columns. Using an attribute sketch greatly improves the functionality of the filter at a modest cost in space. A CCF differs from both cuckoo hashes and filters in that keys may not be unique in the CCF and require techniques to handle duplicates. This is since a CCF must store information about key, attribute pairs and not just a key. While existing cuckoo hash tables and filters only support inserting a small number of duplicated keys, we introduce a chaining technique that allows storage of additional duplicates.

The CCF supports two useful operations. Given an item \( x \) and predicate \( P \), it tests if \( x \in S_P \), in other words, if there is a matching row in the input data. Given just a predicate \( P \), some variations of the CCF return a cuckoo filter for the set \( S_P \). Like other approximate set membership sketches, it returns no false negatives.

2 RELATED WORK
A number of data sketches address the approximate set membership (ASM) problem. These include Bloom filters [6], d-left counting Bloom filters [7], quotient filters [4], and cuckoo filters [20]. Several variants [9, 21, 24, 38, 44, 52, 60] improve them through using cache awareness or compression, or add functionality to support counting. These sketches significantly reduce the amount of space required in practical regimes. Theoretical work [48] proposes a sketch that reduces the asymptotic space usage to the information theoretic minimum. In all these cases, the structures only address simple set membership queries with no notion of predicates.

These structures are used in databases to speed up a number of operations. Most related to our work are the join filters used in Oracle [15, 37], Microsoft SQL Server [22], Informix XPS [61], and SAP ASE [54]. Given a set of dimension tables and a large fact table,
joini Iters construct Bloomfi Iters when scanning the dimension tables and applying any predicates on them. Thesefi Iters effectively push down predicates on dimension tables to the fact table scan and significantly reduce its output. Our work allows suchfi Iters to be precomputed and stored. This allows the fi Iters to be applied to dimension tables on the build side of the join. This can result in, for example, smaller hash tables which do not spill data to disk.

In databases, ASM sketches have been particularly useful for distributed join processing [8, 39, 42, 45] by reducing the number of tuples that must be loaded or sent across a network. They have also been used in join size estimation [46] and computing approximate join results [53]. ASM sketches are extensively used in log structured merge (LSM) tree based key-value stores [2, 3, 19, 23, 55] and in testing if query results are in a cache [47]. Outside of databases, approximate set membership sketches have a wide range of applications, particularly in networking [11, 58].

Also related are methods that sketch attribute columns. Column sketches [27] create small and hardware optimized data sketches that speed up scans involving a predicate. Bloom indexes in Postgres [25] similarly use Bloomfi Iters on rows. A scan on the small sketch locates a subset of the full data that must be read.

For predicate pushdown methods, several methods search for hard constraints between columns in order to translate predicates on one column to a predicate on the join key. Such hard constraints include functional dependencies [16, 26], equivalent columns [12, 28, 31], and magic sets [17, 56, 57] and can be costly to search for. Recent work [30] infers cheaply computed zone maps [62] to map predicates on one column to coarsened range predicates on the join key at the partition level. However, each of these methods require strong interdependence of columns that is also of a particular form, and this dependence often does not exist in the data [56].

3 FILTERS IN JOIN PROCESSING

ASM sketches, such as Bloomfi Iters, aid in reducing the cost of processing joins and semi-joins. Byfi Itering out tuples early in query execution, they can reduce the size of intermediate results and their associated downstream processing costs.

In a query plan with joins, each pairwise join is executed byfi first building a temporary structure such as a hash table or sorted array that 1) contains all tuples from one side or table of the join that potentially affect thei nal output and 2) can be efficiently accessed by the join key. Then tuples on the other side can be efficiently matched and joined to entries in the temporary structure.

ASM sketches canfi Iter out additional tuples at the veryfi rst step when scanning a table to build the temporary structure, thus potentially reducing the number of tuples in all downstream operations as well. This has several benefits. In memory limited scenarios, this can convert a Grace hash join that spills tuples to disk to a simple hash join that processes all tuples in memory [35]. In multi-way joins, reducing the size of intermediate pairwise join results also reduces downstream processing costs. In distributed settings, this can reduce the number of tuples sent across a network. In columnar stores, this can also avoid row stitching costs for output tuples [37].

To see how ASMfi Iters reduce the number of tuples created, consider the following example that returns the production companies that have used dedicated production designers in a TV episode using the IMDB dataset.

This query is a star join among three large tables with predicates applied to all three tables. Figure 2 shows a typical join tree for the query, executing pairwise joins (mc = ci) = t, and shows the number of tuples created in the 1) initial scan of mc, 2) the results of the fi rst pairwise join, and 3) the fi nal output of the entire join.

A typical execution engine can only apply predicates specifi ed on mc in the initial scan on mc to build a hash table. Information in tables t and ci is not used. Likewise, the fi rst pairwise join mc = ci cannot use information in t. ASM sketches allow all three tables to be utilized at the fi rst step of scanning mc and reduces the number of created tuples. The reduction naturally propagates to downstream operations. An ASM sketch on movie_id in ci, and likewise for t, can approximately restrict the hash table from the initial scan to tuples with movie_id in the intersection of all three tables. This can be benefi cial when the intersection is much smaller than the table being scanned.

However, on the IMDB dataset described in Section 8 this intersection is large as all tables contain almost all movie_id’s when no predicates are applied. Figure 1 shows a traditional ASM sketch provides little benefi t since the intersection contains nearly the same movie id as mc alone, but if predicates can be applied to all tables, the size of the intersection can be greatly reduced. An ASM sketch that supports predicatefi Iters can restrict tuples in the initial table scan on mc to be approximately those in this small intersection. Figure 2 shows this ensures downstream operations also generate fewer tuples. Replicating this functionality with regular ASM sketches requires additional table scans that apply predicates, build ASM sketches, and, in distributed systems, transfer the sketches.

4 PRELIMINARIES

Our methods modify cuckoo fi Iters and hashes by adding sketched information about attribute values to each key. Since a single key can have multiple associated attribute vectors, multiset representations of the key are of particular importance. Before describing our methods, wefi rst review cuckoo hashing techniques.
Table 1: Cuckoo hash table

| Symbol | Meaning                  |
|--------|--------------------------|
| $h$    | Hash function            |
| $(k,a)$| Query for key $k$ and attributes $a$ |
| $\ell,\ell'$| Bucket and alternate bucket for $k$ |
| $\kappa$| Key fingerprint          |
| $a$    | Attribute fingerprint vector |
| $\beta$| Load factor of hash table |
| $\oplus$| XOR operation             |
| $b$    | Number of entries per bucket |
| $d$    | max # of duplicate keys in bucket pair |
| $H$    | Cuckoo hash table        |
| $l_{\max}$| Maximum chain length   |
| $H_{e_i}$, $H_{e_i}$| Set of entries, or entry $i$, in bucket $\ell$ |
| $m$    | Number of buckets in table |
| $(k,a) \in H, k \in H$| CCF returns true for query $(k,a)$ |

4.1 Cuckoo hashing

Cuckoo hash tables [49] are a form of open addressing hash table. Such a hash table is arranged as a fixed size array of entries to avoid the overhead of storing pointers. Since hashed locations cannot grow in size, collision resolution techniques must use space among the remaining empty entries. This becomes increasingly difficult as the table reduces the space overhead of empty entries and the proportion of filled entries. For example, linear probing has an expected query and insertion cost of $O(1 + 1/(1 - \beta)^2)$, where $\beta$ is the load factor of the table. According to [20], an optimally sized table empirically achieves $\beta \approx 95\%$ when $b = 4$. Compared to Bloom filters which require $\approx 1.44\log_2 1/\rho + 3/\beta$ bits per item to achieve a desired FPR of $\rho$, where $\beta$ is the load factor of the table. To check if a key $k$ exists, the items in its 2 bucket choices are examined for the key. The bucket search failure probability is $(1 - \beta)^2$. This is a small number of bits. It does not store the full key or any value associated with the key.

Unlike collision resolution techniques where the locations of items in the hash table are immutable after insertion, cuckoo hashing can relocate items when there are collisions at insertion time. By resolving collisions at insertion time, the number of buckets probed at query time can be reduced to two while still being able to achieve a high load factor on the table. Furthermore, cuckoo hash tables have $O(1)$ amortized expected insertion time. Cuckoo hash tables are typically arranged in a tiered fashion so that an item is first hashed to one of $m$ candidate buckets. Each bucket contains $b$ entries in which data can be stored.

The fundamental operations in cuckoo hashing are the insertion and kick out operations. When inserting an item, value pair $(k,v)$, cuckoo hashing hashes the key to a choice of two possible buckets and sets of entries, or entry $i$, in bucket $\ell$. The remaining empty entries. This becomes increasingly difficult as the table reduces the space overhead of empty entries and the proportion of filled entries. For example, linear probing has an expected query and insertion cost of $O(1 + 1/(1 - \beta)^2)$, where $\beta$ is the load factor of the table. According to [20], an optimally sized table empirically achieves $\beta \approx 95\%$ when $b = 4$. Compared to Bloom filters which require $\approx 1.44\log_2 1/\rho + 3/\beta$ bits per item to achieve a desired FPR of $\rho$, where $\beta$ is the load factor of the table. To check if a key $k$ exists, the items in its 2 bucket choices are examined for the key. The bucket search failure probability is $(1 - \beta)^2$. This is a small number of bits. It does not store the full key or any value associated with the key.

Unlike collision resolution techniques where the locations of items in the hash table are immutable after insertion, cuckoo hashing can relocate items when there are collisions at insertion time. By resolving collisions at insertion time, the number of buckets probed at query time can be reduced to two while still being able to achieve a high load factor on the table. Furthermore, cuckoo hash tables have $O(1)$ amortized expected insertion time. Cuckoo hash tables are typically arranged in a tiered fashion so that an item is first hashed to one of $m$ candidate buckets. Each bucket contains $b$ entries in which data can be stored.

The fundamental operations in cuckoo hashing are the insertion and kick out operations. When inserting an item, value pair $(k,v)$, cuckoo hashing hashes the key to a choice of two possible buckets and sets of entries, or entry $i$, in bucket $\ell$. The remaining empty entries. This becomes increasingly difficult as the table reduces the space overhead of empty entries and the proportion of filled entries. For example, linear probing has an expected query and insertion cost of $O(1 + 1/(1 - \beta)^2)$, where $\beta$ is the load factor of the table. According to [20], an optimally sized table empirically achieves $\beta \approx 95\%$ when $b = 4$. Compared to Bloom filters which require $\approx 1.44\log_2 1/\rho + 3/\beta$ bits per item to achieve a desired FPR of $\rho$, where $\beta$ is the load factor of the table. To check if a key $k$ exists, the items in its 2 bucket choices are examined for the key. The bucket search failure probability is $(1 - \beta)^2$. This is a small number of bits. It does not store the full key or any value associated with the key.

4.2 Cuckoo filters

Cuckoo filters [29] adapt cuckoo hash tables for key-only approximate set membership queries rather than key-value queries. There are two primary differences from cuckoo hash tables. First, to save space, a cuckoo filter stores only a small fingerprint $\kappa$. This is a hash of the key $k$ into a small number of bits. It does not store the full key nor any value associated with the key. Second, it uses partial-key cuckoo hashing where the alternate bucket $\ell' = \ell \oplus h(k)$ is determined only by the bucket $\ell$ and the fingerprint. Unlike cuckoo hashing which requires the full key, the alternate bucket can be computed using just the key fingerprint stored in the sketch.

To check if a key $k$ exists, the items in its 2 bucket choices are checked for a matching fingerprint. Trivially, if $k$ was inserted before then the filter will fail a match, so there are no false negatives. If $k$ was not previously inserted, there is some probability of a false match due to random hash collisions on the fingerprint.

For a typical setting where the number of entries per bucket is $b = 4$, an optimally sized cuckoo filter requires approximately $(\log_2 1/\rho + 3)/\beta$ bits per item to achieve a desired FPR of $\rho$, where $\beta$ is the load factor of the table. According to [20], an optimally sized filter empirically achieves $\beta \approx 95\%$ when $b = 4$. Compared to Bloom filters which require $\approx 1.44\log_2 1/\rho$ bits per item to achieve the same FPR, an optimally sized cuckoo filter requires fewer bits per item when the desired FPR $\rho < 0.35\%$. Additional semi-sorting optimizations can further reduce the space so that they are smaller than an optimally sized Bloom filter when the target FPR $\rho < 2.5\%$. 

4.3 Multisets

A conditional ASMs sketch that supports predicates effectively stores a multiset rather than a set. It contains information for a set of key-attribute pairs $(k,a)$. Since the key is not necessarily unique, the collection of search keys $k$ forms a multiset. Existing cuckoo filters have limited support for multisets by allowing duplicate copies of a key fingerprint to be inserted. While this makes cuckoo filters more flexible than Bloom filters as they support deletions by removing a copy of a key fingerprint at most $2B$ copies of a key can be inserted into its two buckets. Furthermore, there are no theoretical guarantees that a high load factor can be obtained. Empirically, we see they cannot. Figure 3 shows the load factor for a plain cuckoo filter decreases dramatically when there are duplicate keys, and insertions into the filter can fail almost immediately when the distribution of duplicate keys is highly skewed. Cuckoo filters can also be extended by adding a counter to each entry, but this approach cannot store attribute information.
We now introduce the Conditional Cuckoo Filter (CCF). These "fingerprints, a Bloom filter, or a mixture of the two. Next, we develop two novel extensions to cuckoo filters and hashing in order to allow for more duplicate keys. We introduce a chaining mechanism which allows the CCF to use more buckets as more duplicates are encountered and a method to switch from fingerprint vectors to Bloom filters.

5.1 Attribute fingerprint vectors

The simplest method to sketch an attribute vector hashes each attribute value into a small number of bits $s$, say 4 or 8, to construct a vector of attribute fingerprints that is significantly smaller than the original data. Despite their small size, these attribute sketches can be effective in applications. For example, in join processing, the FPR does not need to be extremely low to be effective. The output size after applying an equality predicate to an intermediate scan is

$$E_{output} = M_{true} + E_{FPR} \cdot (M_{original} - M_{true})$$

where $M_{original}$ is the total number of tuples in the original table, $M_{true}$ is the number of true matching tuples, and $M_{output}$ is the number of tuples that match according to the CCF. Thus, when

$$\frac{M_{true}}{M_{output}} \approx 0$$

a relatively poor FPR of just 10% reduces the number of tuples produced by a factor of nearly 10. Furthermore, if more than one predicate is applied, the reduction can be multiplicatively amplified. Similarly, there is little reason to target a FPR much smaller than $M_{true}/M_{original}$, as this unavoidable cost becomes the dominant cost in processing.

The resulting CCF will thus have a very low FPR if a key is absent from the set and a higher FPR if the key exists but there is no matching attribute. We use $a$, $\alpha$ to denote an attribute vector and its fingerprint vector. When using attribute fingerprint vectors, the underlying cuckoo hash table must be modified to handle duplicated keys with unique attribute fingerprints.

5.2 Bloom filter attribute sketches

A second choice represents attributes with a Bloom filter. Each (attribute name, value) pair is inserted into a small Bloom filter for each entry. Algorithm 1 summarizes the procedure for querying the filter. The only difference from a regular cuckoo filter query is the additional check to verify that the query predicate matches the attribute sketch, namely all attributes in the query predicate are found in the Bloom filter. A CCF using Bloom attribute sketches can also support queries that only specify a predicate and not a key. It returns a cuckoo filter which a downstream process can use to check the existence of a key. To do this, simply erase all entries where the predicate does not match and return the resulting array of key fingerprints. This is summarized in Algorithm 2.

\begin{algorithm}
\caption{PredicateQuery(a)}
\textbf{Function:} \text{PredicateQuery(a)}
\textbf{Input:} $a$ (attribute vector)
\textbf{Output:} $H'$ (output sketch)

$m, b \leftarrow \text{Dimension}(H)$
$H' \leftarrow \text{new CuckooFilter}(m, b)$
\textbf{for} $\ell \leftarrow 1, \ldots, m$ \textbf{do}
\textbf{for} $i \leftarrow 1, \ldots, b$ \textbf{do}
$(\kappa, \alpha) \leftarrow H_{\ell, i}$
\textbf{if} $a$ matches $\alpha$ \textbf{then} // check for predicate matches
$H'_{\ell, i} = \kappa$
\textbf{return} $H'$
\end{algorithm}

Using a Bloom filter attribute sketch has mixed effects on the size of the sketch and the FPR. First, note that the occupied entries in the sketch are exactly the same as those of a cuckoo filter. Thus, appropriately sized filters are theoretically guaranteed [18] to obtain high load factors with high probability. For our other methods, we only have empirical results showing high load factors are obtained. However, a Bloom filter is less bit efficient than fingerprint
vector. An optimized Bloom filter requires $\approx 1.44 \log_2(1/p)$ bits per attribute to achieve an FPR of $p$ versus $\log_2(1/p)$ for a fingerprint vector. This inefficiency is exacerbated since it is not possible to choose optimal parameters for the Bloom filter. The optimal choice for the number of hash functions to use depends on the number of distinct (attribute, value) pairs that will be added to the filter. These are not known in advance and can vary greatly across keys. Second, when multiple rows of data share the same key but have different attribute vectors, a Bloom filter attribute sketch does not encode which attribute values occur together in the same row. If two rows contain the query key with attribute vectors $(a_1, a_2), (a'_1, a'_2)$, the predicate $A_1 = a_1 \land A_2 = a'_2$ has no matching row, but a Bloom filter attribute sketch is guaranteed to return a false positive. This remains true if the Bloom filter was replaced with any other ASM sketch. On the other hand, if all queries contain only a single equality predicate, then fingerprint vectors can unnecessarily store a single attribute value $A_1 = a_1$ multiple times when the second attribute $A_2$ is varying. A Bloom filter only encodes it once.

5.3 CCF Multiset representations

One advantage of attribute fingerprint vectors is the ability to store co-occurrence information. This comes at the cost of a single key occupying multiple entries in the CCF if it has multiple attribute vectors that differ in any way. The ability to handle a relatively large number of duplicates is an important ability that normal cuckoo hash tables lack. A key’s two buckets contain at most $2b$ entries, and inserting any more copies of a key is guaranteed to fail. This is problematic as many data sets have highly skewed key distributions. Existing ASM sketches that support duplicates [20, 50] either fail after the buckets are full or only maintain a counter that cannot store attribute information.

We present two strategies to address this and ensure no false negatives are produced. One converts attribute fingerprint vectors to Bloom filters when too many duplicates are encountered. We refer to this approach as Mixed. The second is a form of chaining that allows a key to utilize more than 2 buckets. In both cases, we allow a maximum of $d$ duplicated key fingerprint vectors per bucket pair. If an insertion for key $k$ is on a bucket pair already containing $d$ copies of the fingerprint $\kappa$, then either the $d$ copies are converted to a Bloom filter attribute sketch or additional bucket pairs are considered via the chaining procedure. We are not aware of prior work on chaining multiple buckets in cuckoo filter iterators, though [60] describes entire cuckoo filter tables. Chaining tables increases the total capacity of an iterator but forces the sketch to be large if there is skew in the key distribution. We note that although previous works [7, 21, 50] address multiset representations, they do so via counters that cannot store predicate information.

5.4 Bloom filter conversion

Consider a key $k$ and attribute vector $a$ to be inserted in the bucket pair $\ell, \ell'$. Let $|k|, |a|$ denote the size of the key and fingerprint vectors and $n_a$ denote the number of attributes. Suppose there are already $d$ copies of the fingerprint $\kappa$ in the $2b$ entries of $\ell, \ell'$. Bloom filter conversion takes the $d|a|$ bits currently stored in fingerprint vectors and constructs a single Bloom filter in their stead. Each sketch entry also requires an additional bit to track if it contains a Bloom filter or fingerprint vector.

This conversion operation has the advantage that it can never fail. However, it adds complexity in storing a Bloom filter among $d$ entries and maintaining it whenever a bucket’s entry is kicked into the alternate bucket. It has the same advantages and disadvantages outlined in Section 5.2 when directly using a Bloom filter attribute sketch but with two main differences. It has a further disadvantage in that hash collisions can be introduced both from hashing attribute values into fingerprints and from inserting fingerprints into the Bloom filter. Directly using a Bloom filter only introduces collisions from the latter. It has an advantage in that the Bloom filter parameters can be chosen more easily since the minimum number of duplicates is $d$. The Bloom filter parameters do not need to be optimized to also handle rows with a unique key.

The storage of the entries can be further optimized to avoid storing the same key fingerprint multiple times. Instead, each bucket can store a single copy of the key fingerprint along with the number of entries the Bloom filter attribute sketch occupies in that bucket. If the other entries sharing the same key fingerprint are stored contiguously, then the Bloom filter can be successfully reconstructed. In this case, the required number of bits to store fingerprint and counts is $2(|k| + \log_2 d)$ and the number of bits in $d$ entries is $d(|k| + |a| + 1)$.

We choose the number of hash functions to be approximately the optimal number assuming there are $(d + 1) \cdot |a|$ unique attribute (name, value) pairs added to the filter. Let $|B|$ be the number of bits available to the Bloom filter.

\[
\# \text{hashes} = \frac{|B|}{(d + 1) \cdot |a|} \log 2
\]

(2)

\[
|B| = \frac{|a| \cdot d}{d + 1} \log 2 \quad \text{if } |a| > |k|
\]

(3)

Algorithm 3 summarizes the method to convert attribute fingerprint vectors to a Bloom filter. Figure 4 shows how fingerprint and attributes are stored for one bucket in the CCF.

Algorithm 3 BloomConversion($H_\ell$ and $H_{\ell'}$ by fingerprint $\ell, \ell'$, $\kappa$)

Sort entries in $H_\ell$ and $H_{\ell'}$ by fingerprint $\ell, \ell'$. Let $s$ be the size of the fingerprint $\kappa$. If $i = \ell, \ell'$, $s$ is the size of single entry.

$$\text{numHash} \leftarrow \frac{|a|}{|a| \cdot d} \log 2$$

$$\text{totalBits} \leftarrow ds - 2(|k| + \log_2 d))$$

$B \leftarrow \text{Bloom(numHash, totalBits)}$

for $(\kappa', a') \in H_\ell \cup H_{\ell'} \cup \{(k, a)\}$ do

if $\kappa' = \kappa$ then

for $j = 1, \ldots, |a'|$ do

Insert $(\ell, a'_j)$ into $B$

end

end

for $j = 1, \ldots, |a'|$ do

Insert $(\ell, a'_j)$ into $B$

end

5.5 Chaining

Chaining introduces additional bucket pairs whenever an insertion would violate the constraint that at most $d$ copies of a key fingerprint $\kappa$ are in a bucket pair $\ell, \ell'$. A second bucket pair is determined.
Figure 4: Fields in one CCF bucket with 2 entries per bucket.
In Bloom filter conversion, the grey area may be repurposed as a single Bloom filter, and one additional bit indicates if the conversion is performed.

Figure 5: Illustration of chaining procedure

by hashing the bucket pair and fingerprint, $\tilde{t} := h(\min(t, t'), \kappa)$ and $\tilde{t'} := t \oplus h(\kappa)$. This can be inductively applied to generate a sequence of bucket pairs, which we cap at $L_{\text{max}}$ pairs. By using only information about the original key $k$ stored in the CCF, namely the unordered bucket pair $(t, t')$ and fingerprint $\kappa$, this procedure generates the same sequence of bucket pairs during a cuckoo kickout, that has no access to $k$, as a cuckoo insertion or query with access to $k$. The min can also be replaced by any symmetric function.

A query or insertion searches for a match or until the first non-full bucket where a full bucket pair is one containing $d$ copies of the fingerprint $\kappa$. If all $L_{\text{max}}$ bucket pairs are checked and full, then an insertion discards the insertion item, and a query returns true regardless of the predicate to avoid false positives. The query algorithm is summarized in Algorithm 5. An insertion is illustrated in Figure 5 and summarized in Algorithm 4. Correctness is proven by the following lemmas and theorem.

**Lemma 1.** Consider a CCF with chaining. Given any key fingerprint $\kappa$ and bucket $t$, let $t' = t \oplus h(\kappa)$ be the alternative bucket. The total number $c_k(t)$ of copies of $k$ in buckets $t, t'$ after $t$ insertions is increasing and capped by the parameter $d$.

**Proof.** We prove this by induction. This trivially holds in the base case where the CCF is empty. $c_k(t)$ can only decrease if an entry containing $k$ is kicked out in a cuckoo kick operation. Assume WLOG that a copy of $k$ in bucket $t$ was kicked out at the $t + 1$ insertion. Suppose $c_k(t) < d$. After the kickout there are still $< d$ copies of $k$ in $t, t'$ and the kicked out copy must be reinserted in these buckets. Thus $d \geq c_k(t) + 1 \geq c_k(t + 1) \geq c_k(t)$. Now suppose $c_k(t) = d$. If the new fingerprint $k' \neq k$, then after the kickout there are fewer than $d$ copies of $k$ and it must be reinserted into the same bucket pair. Otherwise $k' = k$, and the insertion must choose a new bucket pair. Since $t' = t \oplus h(\kappa)$, any new bucket pair must either be identical to $(t, t')$ and another bucket pair must be generated, or it is disjoint. In both cases, the bucket pair $t, t'$ is unaffected so $c_k(t) = c_k(t + 1) = d$.

**Lemma 2.** Let $(k, \alpha)$ be a key-attribute vector pair with corresponding fingerprints $(\kappa, \alpha)$ and $H$ be a CCF with chaining. There is a fixed sequence of buckets $t_1, t_2, \ldots t_n$ determined by $k$ with $n \leq 2L_{\text{max}}$ that it can be inserted into. Furthermore, if at any time it is in $t_i$, then all pairs $t_{j+}, t_{j-1}$ with $2j - 1 < i$ contain $d$ copies of $\kappa$.

**Proof.** The sequence is recursively defined by $t_i = h(t_{j}(k), t_{j+1} = t_{j+1} @ h(\kappa)$ and $t_{j+1} = h_{(\min(t_{j+1}, t_{j+1}), \kappa)}$. It ends if there is a cycle or the maximum chain length of $L_{\text{max}}$ bucket pairs is reached. This sequence is deterministic since $t$ and $\kappa$ are deterministic functions of $k$. A key-attribute pair is inserted in the first bucket pair in the chain with fewer than $d$ copies of $k$. A reinsertion after a kickout operation from bucket pair $t_{j}, t_{j+1}$ does the same starting from $t_{j}$. Thus, each pair $t_{j}, t_{j+1}$ with $2j - 1 < i$ contain $d$ copies of $\kappa$. Lemma 1 shows there are exactly $d$ copies.

**Theorem 3.** A CCF with chaining and no failed insertions returns no false negatives.

**Proof.** Given a key $k$ and predicate $P$, let $k, \alpha$ be some matching row with fingerprints $\kappa, \alpha$. If no rows with key $k$ are discarded, then by Lemma 2, all bucket pairs before the actual location corresponding to $k$, a contain exactly $d$ copies of $k$’s fingerprint $\kappa$. Otherwise, by Lemma 1, all $2L_{\text{max}}$ buckets in the bucket chain for $k$ contain $d$ copies of $\kappa$. Since a query for $k, \alpha$ will continue searching bucket pairs along the chain for $k$ when there are $d$ copies of $k$’s fingerprint $\kappa$ in the bucket pair, it will either find a matching entry or reach the maximum chain length. Both conditions return true.

Note that generating alternative bucket pairs may create a cycle of pairs. In this case, the insertion procedure will not fail but will not generate $L_{\text{max}}$ unique bucket pairs. Such cycles can be detected using Floyd’s cycle detection algorithm and the chain can be extended using the same chaining procedure with a different hash function for each cycle detected but taking the minimum bucket in the entire chain rather than bucket pair.

**Algorithm 4** ChainInsert($\tilde{t}, \kappa, \alpha, L_{\text{max}}$)

$t' = t \oplus h(\kappa)$

if $(\kappa, \alpha) \in H_\ell \cup H_\ell'$ then
   return Success

else if $[(\kappa', \alpha') \in H_\ell \cup H_\ell' : \kappa' = \kappa]) = \text{MaxDupes}$ then
   $\tilde{t} = h(\min(t, t'), \kappa)$
   ChainInsert($\tilde{t}, \kappa, \alpha, L_{\text{max}} - 1$)
else if $H_\ell$ is not full then
   Insert $(\kappa, \alpha)$ into $H_\ell$
   return Success

for $i \leftarrow 1$ to MaxKicks do
   if $H_\ell$ is not full then
      Insert $(\kappa, \alpha)$ into $H_\ell$
      return Success

Pick random entry $i \leq b$
Swap$(\kappa, \alpha)$ with $H_\ell[i]$
$t' \leftarrow t' \oplus h(\kappa)$
return Terminated

Like Bloom filter attribute sketches, the chaining method can support predicate only queries. However, it cannot simply erase entries with non-matching attribute values. This could introduce gaps in a chain where some bucket pair does not contain $d$ copies of a fingerprint. A query could stop probing bucket pairs early and yield false negatives. Instead, the sketch must keep the key fingerprint and add a bit to mark the entry as non-matching.
We show that all the variations of the CCF are governed by a bound on the FPR of different CCFs. The accuracy of CCF's and other approximate set membership checks can be measured by the false positive rate (FPR). Unlike regular cuckoo filters, the FPR for CCF’s is not a constant. Queries can result in false positives due to spurious matches on the key fingerprint, on the attribute sketch, or both. Because of this, the FPR depends on the number of non-empty entries, this gives a bound on the FPR for key only queries using Bloom attribute sketches as follows:

\[ P_{\text{Bloom}}(P \in H[k]) = \rho(z_k)^u \]  

(6)

where \( \rho(z_k) \) is the FPR of a Bloom filter with \( z_k \) entries added into it. The FPR thus decreases exponentially quickly as there are more unmatched attributes. Also note that when query predicates on different attributes match on different rows, \( u \) can equal 0, and the FPR can be 1 as described in Section 5.2.

When using attribute fingerprints, the FPR depends on the number of duplicate keys inserted, the attribute fingerprint length \( |\alpha| \), and the number of predicates that are false on each row. The probability the \( i \)th entry in the CCF with key fingerprint \( \kappa \), denoted by \( H[k]_i \), triggers a false positive is \( P(\alpha \in H[k]_i) = \rho(\alpha) \) where \( \rho(\alpha) \) is the number of non-matching predicates on the input data row corresponding to entry \( H[k]_i \) and \( \rho = 2^{-|\alpha|} \) is the probability of a collision in the fingerprint. Summing over the false positive probabilities for the entries gives an upper bound on the FPR:

\[ P_{\text{chained}}(P \in H[k]) \leq d_{\text{max}} E^2 2^{-|\alpha|} \tilde{U} \]  

(7)

since there are at most \( d_{\text{max}} \) entries for key \( k \). Here, \( \tilde{U} \) denotes a randomly chosen \( \tilde{u}_i \).

For Bloom conversion, the FPR depends on whether the entry for the key \( k \) has been converted to a Bloom filter. Although the formulae are upper bounds on expected FPR, Figure 6 shows they are reasonably good predictors of actual FPR.
Figure 6: The bounds on the expected FPR are good predictors of the actual FPR when using attribute fingerprints.

6.3 Size and parameter choice

Conditional cuckoo filters have more parameters than regular cuckoo filters. Cuckoo filters only require setting the number of buckets $m$ and the number of entries per bucket $b$. CCF’s can additionally require setting the maximum number $d$ of duplicates per bucket pair, $L_{\text{max}}$ the maximum chain length, and any additional parameters required by the attribute sketch.

We derive an upper bound on the number of non-empty entries and experimentally show the attainable load factor is not sensitive to the underlying data. Together these can be used to size the sketch. These bounds and constants depend on parameters $d$ and $L_{\text{max}}$ which affect the number of duplicates stored in the sketch.

Denote the total number of distinct keys by $n_k$ and the number of non-zero entries in a CCF by $Z^c$. Since a CCF with Bloom attribute sketches has the same non-empty entries as a regular cuckoo filter, the number of non-zero entries can be upper bounded by $n_k$. For other cases, let $r_k$ be the number of duplicates for key $k$ that have distinct attribute values. Bloom filter conversion will allocate a maximum of $\max(d,r_k)$ entries for that key. Let $A = r_X$ for a randomly chosen key $X$. Then the expected number of used entries is bounded by $E Z^c \leq n_k E \min\{A,d\}$. Similarly, a CCF with chaining uses at most $dL_{\text{max}}$ entries for a single key, so the expected number of entries used is bounded by $E Z^c \leq n_k E \min\{A,dL_{\text{max}}\}$. These sizes are summarized in Table 1. Figure 7 shows the bound on the number of entries closely matches the actual number needed.

From experiments on chaining, we find a reasonable rule of thumb sets the number of entries per bucket to $b = 2d$. This way, at least 4 distinct keys can be stored in a bucket pair $\ell,\ell'$. Typically more keys can be stored since only 1 key fingerprint can be scanned. Figure 3 shows that a setting of $b = 4$ that is typical for cuckoo filters achieves a load factor of around $\beta \approx 75\%$ regardless of the number of duplicate keys. A slightly larger value of $b = 6$ achieves a load factor close to $\beta \approx 87\%$ even when there are many duplicates. An appropriate estimate for the required size for a CCF is $m \cdot b \approx E Z^c/\beta$.

When the sketch is optimally sized, Figure 8 shows that lower settings for $d$ tend to achieve better use of bits. This is primarily due to smaller values of $d$ yielding higher load factors. Given an indexed setting for the number of bins $m$ and entries per bin $b$, we found the best $d$ is the largest which all insertions pass. We chose $d = 3$ which provides small bucket sizes and a good load factor.

6.4 Attribute sketch parameters

Figure 11 shows the performance of different CCF’s under various parameter choices. Generally, we also found increasing the attribute sketch size more beneficial than increasing the key fingerprint size. Figure 6 shows that at small attribute sketch sizes, the false positives are primarily due to bad matches on the attribute sketches. In this case, a false positive $k$, is ascribed to a key if the key is not in the sketch, in other words, $k \not\in H$. Otherwise, it is easily ascribed to the attribute. We also found small values for the number of hash functions used by Bloom filters to be preferable as it does not become ill with one too quickly.

7 RANGE QUERIES

Conditional Cuckoo filters can be extended to support range predicates. Given a column with a range predicate, one simple method is to bin the column into a small number of bins. A range predicate can then be converted into a small in-list. The disadvantages of this approach are that the binning process introduces error and that long ranges must check more bins. Since each bin that does not contain a true match can return a false positive, both can increase the FPR. Another method uses a standard approach of using a dyadic expansion over the range $[a_0, b_0]$ of the column. An item $x$ can be represented as a sequence of intervals $[a_1, b_1], \ldots, [a_\eta, b_\eta]$ with exponentially decreasing lengths $b_{i+1} - a_{i+1} = (b_i - a_i)/2$ down to some final granularity $b_\eta - a_\eta$. This requires $\eta$ insertions into a CCF for each item, and a range query likewise requires querying for the existence of up to $\eta$ intervals that cover the range. This is primarily useful when the number of desired intervals is large since $\eta$ has a logarithmic dependency on the number of intervals. We use the simpler binning in our experiments since they use few bins.

8 EXPERIMENTS

We consider two experiments. The first “multiset experiment” uses synthetic data to examine the ability of the chaining procedure to store duplicate keys while obtaining a high load factor. The second “JOB-light” experiment examines the ability of the CCF to reduce output sizes on a join benchmark on real data [33].
8.1 Multiset experiment data and setup

Data: Our experimental results show chaining dramatically improves the ability of a cuckoo filter to handle duplicate keys. We simulated key frequencies using either a truncated Zipf-Mandelbrot distribution or a stream where every key has the same number of duplicates. We do not generate attribute vectors for this experiment. For the Zipf-Mandelbrot distribution with a mass function of the form \( p(x) = \frac{(c + x)^{-\alpha}}{\sum_{x \geq 0} (c + x)^{-\alpha}} \), we fix \( c \) to the Mandelbrot parameter 2.71 and truncate the range to be in \([1, 500]\). For both distributions, we generate multiple datasets, each with a different number of average duplicates per key.

Setup: For CCF parameters we vary the number of entries \( b \) per bucket from 4 to 8 and follow the rule-of-thumb in Section 6.3 in setting the maximum number of duplicates per bucket pair to \( d = 3 \). Figure 8 shows this choice also has close to optimal efficiency. The maximum number of bucket pairs for a key \( L_{\text{max}} = \infty \) is uncapped. For each experimental run, we generate a dataset that is approximately 20% larger than the capacity of the sketch and measure the number of items processed before the first failed insertion and the load factor at that point. A failed insertion is the first time a new key, attribute pair fails to generate a new entry in the sketch. The order of items is randomly permuted. We use the additional cycle detection and resolution method in these experiments. The results are averaged over 20 runs using random salts for the hash functions.

We also measure the space efficiency of the chained cuckoo filter as compared to a theoretical information theoretic. We define the bit efficiency of the filter to measure the efficiency as

\[
\text{Efficiency} := \frac{\text{sketch size in bits}}{n \log_2 \frac{1}{\rho}} \in [1, \infty)
\]

where \( n \) is the total number of keys inserted and \( \rho \) is the FPR. When all keys are distinct, a sketch with an efficiency of 1 cannot be improved since it matches the information theoretic lower bound given by the denominator.

8.2 Multiset results

Figure 3 shows the behavior of a regular multiset cuckoo filter and cuckoo filter with chaining as the number of duplicates per key is varied. The chained cuckoo filter is able to achieve roughly the same load factor regardless of number of duplicates. In contrast, the plain cuckoo filter’s ability to achieve a high load factor quickly decreases. For Zipf-Mandelbrot data, the plain cuckoo hash encounters very few items before it fails. The cuckoo filter quickly encounters more copies of these keys than it can store. When the number of generated duplicates per key equals the maximum allowed by a non-chained cuckoo filter \( d \), the chained and plain cuckoo filters achieve similar load factors. However, the chained cuckoo filter has a slightly worse FPR as it inspects an extra bucket pair.

Taking into account both the load factor and the FPR, Figure 8 shows an optimized chained cuckoo filter obtains a bit efficiency of \( \approx 1.93 \) for the optimal or nearly optimal setting of \( d = 3 \) when the number of entries per bucket is \( b = 6 \). In comparison, set membership sketches only gain modestly better efficiency for the

Figure 8: Small values for \( d \) duplicates per bucket pair generally use space more efficiently. While a very low value of \( d = 2 \) can achieve higher load factors, it can result in higher FPRs that make the sketch less efficient.

loss in efficiency in handling duplicates. A Bloom filter has a bit efficiency of \( \approx 1/\log 2 \approx 1.44 \). Removing a CCF’s chaining functionality to obtain a cuckoo filter without semi-sorting gives a bit efficiency of \( \approx 1.53 \) when the FPR is 1%.

8.3 JOB-light experiment

Our second set of experiments evaluate the efficacy of CCF’s for join processing on a real world dataset. We evaluated the CCF’s ability to reduce the output of scan operators using the JOB-light workload [34] derived from the Join Order Benchmark (JOB) [40] used to evaluate the quality of query optimizers.

Given a query and a table in that query, we wish to determine the minimum sized output that a scan operator on that table can provide and compare it to an operator that only applies the predicates on that table. The minimum size output is produced by converting joins of this base table to other tables to semi-joins, which only check if the key exists in the other tables after applying predicates. To execute a semi-join using a CCF, for a row in the base table with key \( k \), each joining table’s CCF is queried for \((k, P)\) where \( P \) is the predicate from the query for that table. We define the Reduction Factor (RF) to be

\[
\text{Reduction Factor} := \frac{M_{\text{semi-join}}}{M_{\text{predicate}}}
\]

where \( M_{\text{semi-join}} \) is the number of base table rows that both match the given predicates on that table and all other CCF’s. The value \( M_{\text{predicate}} \) is the number of rows in the base table that match the predicates on that table only. The concept of reduction factor is similar to predicate selectivity: when reduction factor (selectivity) is 0.0, no rows are selected and when reduction factor (selectivity) is 1.0, all rows are selected. For a distributed system, the reduction factor measures what proportion of tuples are sent over the network, or in non-distributed hash joins, how much smaller the hash table sizes are. We also measure the CCF size, the space cost of the CCF used to achieve these tuple/size reductions.

8.3.1 JOB-light data and workload. The JOB-light workload consists of 70 queries, each joining up to 5 large tables from the IMDB dataset. Among these 70 queries are 237 instances of tables that qualify for matching join keys and predicates in at least one CCF, effectively a semi-join reducer [5].
Table 2: Summary of tables and predicates used in JOB-light workload. High cardinalities in bold.

| Table       | Number of Rows | Predicate Column | Column Cardinality |
|-------------|----------------|------------------|--------------------|
| cast_info   | 36,244,344     | role_id          | 11                 |
| movieCompanies | 2,609,129     | company_id       | 234,997            |
| movieCompanies | 2,609,129     | company_type_id  | 2                  |
| movie_info  | 14,833,720    | info_type_id     | 71                 |
| movie_info_idx | 1,380,035     | info_type_id     | 5                  |
| movie_keyword | 4,523,930     | keyword_id       | 134,170            |
| title       | 2,528,312     | kind_id          | 6                  |
| title       | 2,528,312     | production_year  | 132                |

Table 3: Number of distinct attribute values per key. Equivalently, number of duplicated keys. High Max Dupes in bold.

| Table       | Join Key | Predicate Column | Avg Dupes | Max Dupes |
|-------------|----------|------------------|-----------|-----------|
| cast_info   | movie_id | role_id          | 4.70      | 11        |
| movieCompanies | movie_id | company_id       | 2.14      | 87        |
| movieCompanies | movie_id | company_type_id  | 1.54      | 2         |
| movie_info  | movie_id | info_type_id     | 4.17      | 68        |
| movie_info_idx | movie_id | info_type_id     | 3.00      | 4         |
| movie_keyword | movie_id | keyword_id       | 9.48      | 539       |
| title       | id       | kind_id          | 1.00      | 1         |
| title       | id       | production_year  | 1.00      | 1         |

In this workload, each query involves 2 to 5 of the 6 tables listed in Table 2, and all joins are on the movie identifier. Two tables, movie_companies and title, each contain two predicate columns, thus providing an opportunity to evaluate the workload using a combination of single-attribute and multi-attribute CCF’s. While most predicates are equality predicates, 55 JOB-light queries have inequality predicates on title.production_year which ranges from 1880 to 2019. We applied the simple binning technique of Section 7 and mapped the 132 values to 16 roughly equal-sized intervals and converted inequality predicates to in-lists. In the cases where the scan operation was on the title table, we omitted this binning since the predicate could be evaluated directly.

The relevant IMDB data for the JOB-light workload is summarized in Table 2 and Table 3. These include the predicate columns and their cardinalities. When using 4 bit attribute fingerprint sizes, 4 of the predicates can be considered “high” cardinality with cardinality $>2^4$. Table 3 shows the number of duplicate predicate attribute values per join key. This affects both the sizes of the sketches as well as the FPR. While the Avg Dupes vary from 1.00 to 9.48 distinct duplicates per join key, CCF’s must handle the worst case behavior of Max Dupes which varies from 1 to 539.

8.3.2 JOB-light experiment setup. We evaluated all four CCF methods: Plain (regular cuckoo filter allowing duplicate keys); Chained (CCF w/ chaining); Bloom (CCF w/ Bloom); and Mixed (CCF w/ Bloom conversion); as well as a simple cuckoo filter for a range of attribute sizes, fingerprint sizes, and Bloom attribute sketch sizes.

The following filter parameters were evaluated: attribute fingerprint sizes of $|x| = 4$ or 8 bits, fingerprint sizes of $|x| = 7, 8, 12$ bits, and Bloom filter sizes ranging from 4 to 24. The number of hash functions used in the Bloom filter was either fixed at 2, or was optimized to achieve the lowest FPR under the assumption that 2 attribute vectors are inserted per key. We found the latter setting resulted in uniformly worse FPR’s and omit their results from the rest of this analysis. Like the multiset experiment, the maximum number of duplicate keys per bucket pair was always set to $d = 3$.

The number of buckets and bucket size were independently chosen for each filter based on the analysis in Section 6.3. Given the predicted number of entries, we fit the smallest bucket size which would both result in an acceptable load factor and have high likelihood of successfully inserting all input rows based on the multiset experiment. Note that the predicted number of entries needed can be estimated from the data using a bottom-k [14] or two-level [13] sampling scheme.

8.4 JOB-light results

We compare how the CCF methods perform versus the theoretically best possible RF (Reduction Factor), which makes full use of all predicate information, and against the best existing baseline of a Cuckoo Filter, which throws away information about predicates.

The best possible RF is the Exact Semijoin RF where no false positives are emitted. We also examined the effect of changing the size of fingerprints and attribute sketches on the accuracy of CCF’s.

Large Filters: Figure 9a plots the reduction factor on the y-axis for each of the 237 possible scan operators in the 70 workload queries. The parameters of all filters are the same, having 8-bit attributes, 12-bit fingerprints, and 4 hash functions for Bloomfilters. These filters are ”large” due to the choice of 8 and 12 bits for attribute and fingerprint size respectively, as well as specifying a large Bloom filter sketch. The ordering of these tables on the x-axis is in increasing order of the Exact Semijoin reduction factor; therefore, the reduction factor of all filters should be to the left and above the Exact Semijoin RF line. For large filters, the RF of allfilter methods are fairly close to Exact Semijoin with a few outliers.

Figure 9b also uses large filters, but has a different baseline than in 9a. Here the baseline is based on using large Bloom filters, but only looking up the ifi lters’ join keys, ignoring any other predicates. This baseline behavior is analogous to a regular Cuckoo filter rather than a CCF and represents the current state-of-the-art of pre-built filters. The reduction factor of all filters should be to the right and below the Cuckoo Filter baseline. CCF reduction factors are substantially better than current state-of-the-art pre-built filters. In many cases, where the Cuckoo Filter reduction factor is 1.0, meaning no reduction at all, the CCF’s are in the range of 0.05 – 0.20.

Small Filters: Now we consider “small” filters, using 4-bit attributes, 7-bit fingerprints, and 2 hash functions for Bloom, reducing filter size by more than half. Figures 9c and 9d show the reduction factors by increasing Exact Semijoin and Cuckoo Filter RF baselines respectively, for filters larger than a CCF and represents the current state-of-the-art pre-built filters. The number of non-optimal reduction factors in 9c are more visible. The separation of Bloom CCF reduction factors from Mixed and Chained is particularly noticeable, while small Mixed and Chained RF’s are similar to large filters. Even small CCF’s are substantially better than current state-of-the-art filters.

Plain Filters: Note that none of these filters have results for Plain CCF6 filters as they did not result in reasonably sized filters. The smallest Plain filter was larger than every other CCF and had an inefficient load factor of 35%. Larger attribute fingerprint sizes result in insertion failures for any reasonable filter parameters because
the number of distinct attribute values is too large. For example, as shown in Table 3, movie_keyword.keyword_id has 539 distinct duplicates which would require a minimum bucket size of 270.

8.4.1 JOB-light aggregate results. On aggregate, the reduction factor over all table scans was $\approx 0.28$ using a CCF with chaining and ‘small’ sketches. In contrast, using regular cuckoo filters with no predicate information resulted in a reduction factor of $\approx 0.68$. The best possible reduction factor obtained from performing an exact semi-join was 0.20. Furthermore, half of the difference in reduction factors between the CCF and exact semi-join is explained by the binning on the range predicate on production_year as seen in Figures 10 and 11. If an exact semi-join is performed on data with binned production_year, the optimal reduction factor is 0.24.

Furthermore, using the largest sized CCF, which was a CCF with chaining, 12 bit key fingerprints, and 8 bit attribute fingerprints, the FPR was just 0.8% relative to a semi-join with binned production_year, and the reduction factor was 0.245. The FPR including errors due to binning was 6.1%.

Figure 12 shows that the benefits of CCF’s are compounded as more joins are added. Figure 13 shows the size of a CCF relative to the raw data. Each CCF represents a movie id and the given predicate column. The gains from each CCF can vary significantly based on the underlying data and the number of duplicate keys.
We implemented a single threaded CCF in C++ and ran our experiments using a single Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz core running CentOS Linux 7. We used the Jenkins lookup3 hash function [29] used by the original cuckoo filter paper [20]. All filter methods could process 1 million matches per second.

9 DISCUSSION AND FUTURE WORK

While we focus on the application of CCF’s to join processing, the sketch itself can be seen as a sketch of the entire input table with a hash based index on the key. Furthermore, the chaining technique can also be used to allow regular cuckoo hash tables, which store the full key, to store duplicates. Thus, we believe the sketch and its methods have more general applications beyond join processing.

For example, recent work on Cuckoo Indexes [32] enables data skipping for secondary columns by associating a key for the column to the multiple data partitions containing it. The CCF provides a simple, new solution to the resulting multiset problem.

Another possible application is for access control in information retrieval. Documents in a database may be restricted by subscription status, country, or other factors. However, a typical search engine using inverted indexes may not be able to efficiently restrict results to documents a user has access to. A CCF on document, permission pairs can help efficiently filter the search results. Another possible use is for hierarchical set membership queries. For example, a database may store documents with revision numbers, modification times, or other traits across a collection of servers. A CCF could be used to check if a cache contains a specific document revision, and if not, whether there is any revision already in the cache. It can further aid in resource routing.

Although the sketch is made robust to skew in the key distribution, sizing the sketch is still problematic as it requires predicting the number of led entries in the sketch. Future work to improve this method, as well as many other ASM sketches, includes enabling dynamic adjustment of the size of the sketch.

Furthermore, while we present empirical evidence that on data containing duplicate keys, the CCF with chaining can achieve a load factor comparable to that of a regular cuckoo filter acting on data with no duplicates, we do not have a theoretical proof that this is always the case.

10 CONCLUSIONS

We introduce conditional cuckoo filters, a new sketch for approximate set membership queries which enables equality predicates to be added to queries. This yields at least two significant advantages in join processing. First, it enables filters that are specific to the predicate to be applied to both build and probe sides of a join, not just the probe side. This increases the number of cases where the data structures created on the build side is into main memory. Second, it enables predicate pushdown from one table to all other tables in the transitive closure of the join graph.

We propose, analyze, and evaluate multiple variations of CCF sketches. In particular, we extend cuckoo hash tables using a chaining technique that makes it robust to duplicated keys and allows high load factors to be achieved. All variations reduced the number of rows emitted by a scan operator to close to the optimal number on the workload and did so with substantial space savings. This represents a significant improvement over existing filters that do not support predicates. The properties of the sketches are analyzed which allows practitioners to predict the performance of the sketches and to choose appropriate parameters for them.
