Towards Managing Complex Data Sharing Policies with the Min Mask Sketch

Stephen Smart
School of Computer Science
University of Oklahoma
Norman, Oklahoma
Email: smart@ou.edu

Christan Grant
School of Computer Science
University of Oklahoma
Norman, Oklahoma
Email: smart@ou.edu

***Abstract*—More data is currently being collected and shared by software applications than ever before. In many cases, the user is asked if either all or none of their data can be shared. We hypothesize that in some cases, users would like to share data in more complex ways. In order to implement the sharing of data using more complicated privacy preferences, complex data sharing policies must be used. These complex sharing policies require more space to store than a simple "all or nothing" approach to data sharing. In this paper, we present a new probabilistic data structure, called the Min Mask Sketch, to efficiently store these complex data sharing policies. We describe an implementation for the Min Mask Sketch in PostgreSQL and analyze the practicality and feasibility of using a probabilistic data structure for storing complex data sharing policies.**

***Keywords*-Min Mask Sketch; Probabilistic Data Structure; Sharing Policies; Databases;**

I. Introduction

The storage and management of large data sets is becoming increasingly common. Many applications are continuously recording data about its users and sharing this data to other entities. This leads to data privacy issues and as more data driven applications are coming into existence, these privacy issues are becoming more complex. One approach to handling data privacy when it comes to managing and sharing user data is a simple "all or nothing" approach. In other words, all of the data can be shared or all of it is restricted. This approach works for many applications, but what if the user would like to share a portion of the data being recorded and hide the rest? What if the user would like to share her data in a more complex manner such as dependent on time, location, or a combination of several conditions? These complex policies for data sharing are becoming more practical with the development of more data driven applications and the growth of the underlying network in which these applications communicate, i.e. the Internet of Things.

Complex data sharing policies such as those mentioned above are difficult to implement in a modern database management system. In addition to the overhead added to the development life cycle, complex sharing policies also require more space. Instead of a simple Boolean value representing the "all or nothing" approach to data privacy described above, more bits are needed to represent these policies and every policy could potentially be unique to a single data point in the data set. Many of these data driven applications that are recording user's personal data exist in the mobile application domain, therefore space is an important consideration.

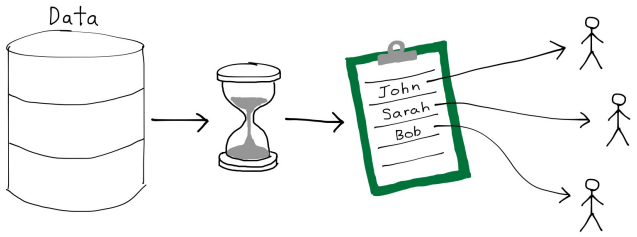

Figure 1: Pictorial depiction of a complex data sharing policy discussed in Appendix II-B.

In this paper, we will describe one approach to improve the space efficiency of storing complex data sharing policies. For example, Figure 1 pictorially describes a complex policy that asks for a set of data records to be shared with a limited set of users for a limited period of time. This approach involves the development and use of a novel probabilistic data structure, that we will call the Min Mask Sketch, to store complex sharing policies in a small amount of space. As with most probabilistic data structures, a small amount of accuracy will be sacrificed in exchange for an increase in space efficiency. One of the most popular probabilistic data structures is the Bloom Filter [1]. The goal of the Bloom Filter is to determine if any given item is a member of a large data set without having to store the entire data set in memory. Over the years, many new probabilistic data structures have been developed that implement a similar approach used in the Bloom Filter but include strategic modifications to answer a different question about the original data. One such data structure is the Count Min Sketch, which not only answers the question of set membership, but additionally can determine the frequency at which a given item exists in the data set. The Min Mask Sketch is a modified version of the Count Min Sketch [2] that can be used to determine a given item's privacy policy.

978-1-5386-1562-1/17 $31.00 © 2017 IEEE
DOI 10.1109/IRI.2017.23

The remaining sections of this paper will be organized as follows:

- Section II discusses the idea of complex data sharing policies in more detail, lists several examples of various sharing policies and describes some of the background work that sparked many of the ideas introduced in this paper.
- Section III introduces an example application and relational schema to illustrate one potential practical application of the Min Mask Sketch approach to storing complex data sharing policies.
- Section IV explains the Min Mask Sketch data structure in detail.
- Section V describes our implementation of the Min Mask Sketch data structure in PostgreSQL 9.6
- Section VI analyzes the feasibility and practical applications of the Min Mask Sketch approach and compares this approach with some alternative methods.
- Section VII summarizes the approach and provides concluding remarks.

II. Complex Sharing Policies

We define complex data sharing as the sharing of data that requires fine grained access control. In other words, each individual data point could be restricted based on a different set of conditions. When data is shared in this way, the standard approach to data privacy does not work. More sophisticated approaches that use complex sharing policies must be used.

That is, given a data set $D = \{r_1, \ldots r_{|D|}\}$, a complex policy C is described by a list of policies $C = \{p_1, \ldots, p_{|C|}\}$ where each simple policy $p_i \rightarrow \Gamma_k$ where $\Gamma_k \subset D$ and $|C| > 1$.

Below, we give examples of various sharing policies. We show how a complex policy can be created from a combination of simple sharing policies. We walk through pictorial representations of both the simple sharing policies and a complex sharing policy. For each policy example, we describe a row level security policy that could be applied in PostgreSQL for enforcing the example policy.

A. Simple Sharing Policy Examples

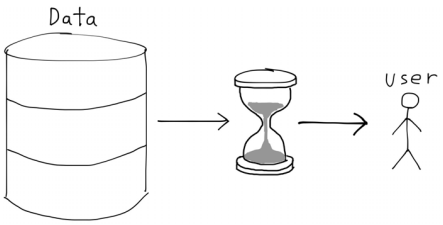

Figure 2: Sharing Data for a Limited Time Period

Sharing Data for a Limited Time Period: The policy shown in Figure 2 describes sharing records from a data set for a limited time period. For example, an owner of a data set would like to share data for 24 hours after which the shared data becomes private. A row level security policy as shown in Figure 3 can be applied in PostgreSQL to enforce this sharing policy.

```
CREATE FUNCTION create_limited_time_policy()
RETURNS void
$BODY$
DECLARE
  end_time timestamp := now() + interval '1 day';
BEGIN
  CREATE POLICY limited_time ON example_data_set
    USING (now() < end_time);
END;
$BODY$
LANGUAGE plpgsql;
```

Figure 3: An example function to create a 24 hour time-limited policy.

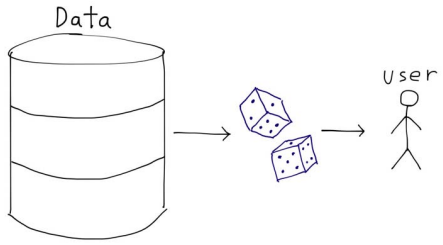

Figure 4: Sharing a Random Sample of Records

Sharing a Random Sample of Records: The policy shown in Figure 4 describes sharing a randomly selected subset of data. For example, an owner of a data set may not want to share all of her data, but instead chooses to share 5% of records selected randomly from the data set. A row level security policy as shown in Figure 5 can be applied in PostgreSQL to enforce this sharing policy.

```
CREATE FUNCTION create_random_sample_policy()
RETURNS void
$BODY$
BEGIN
  CREATE POLICY random_sample ON example_data_set
    USING (random() > 0.95);
END;
$BODY$
LANGUAGE plpgsql;
```

Figure 5: An example function to create a policy that shares a random sample of record.

Sharing a Biased Sample of Records: The policy shown in Figure 6 describes sharing a subset of data according to some bias. For example, a user located in

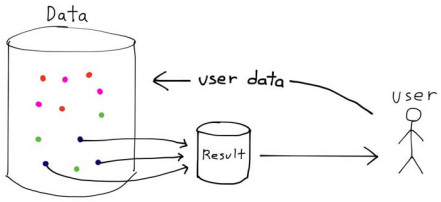

Figure 6: Sharing a Biased Sample of Records

"New York City" queries a weather data set, but only the records relevant to New York City are shared. Assuming the `example_data_set` contains a `location` attribute, a row level security policy as shown in Figure 7 can be applied in PostgreSQL to enforce this sharing policy.

```
CREATE FUNCTION create_biased_sample_policy()
RETURNS void
$BODY$
BEGIN
  CREATE POLICY biased_sample ON example_data_set
    USING (EXISTS
            (SELECT 1
             FROM users u
             WHERE u.user == current_user
               AND u.current_loc == location));
END;
$BODY$
LANGUAGE plpgsql;
```

Figure 7: An example policy to create a location-based sharing policy.

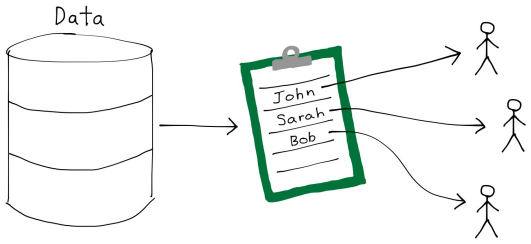

Figure 8: Sharing Data with a Set of Other Users

Sharing Data with a Set of Other Users: The policy shown in Figure 8 describes sharing data to a specific set of users. For example, a Facebook user would only like his/her data to be shared with users on his/her friends list. A row level security policy as shown in Figure 9 can be applied in PostgreSQL to enforce this sharing policy.

Sharing a Limited Number of Records: The policy shown in Figure 10 describes sharing a set number of records from a data set and restricting the rest. For example, an owner of a data set would like to share a maximum of 100 records. A row level security policy as shown in Figure 11 can be applied in PostgreSQL to enforce this sharing policy.

```
CREATE FUNCTION create_set_of_users_policy()
RETURNS void
$BODY$
BEGIN
  CREATE POLICY set_of_users ON example_data_set
    USING (current_user IN
      (SELECT user_name FROM allowed_users));
END;
$BODY$
LANGUAGE plpgsql;
```

Figure 9: An example function to create a sharing policy between users in a data set.

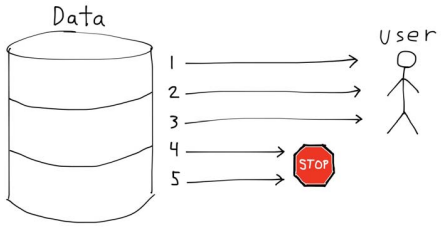

Figure 10: Sharing a limited number of records

In this example, a `limit_reached` function takes two parameters, `table_name` and `limit`, then ensures the number of shared records does not exceed the `limit`.

```
CREATE FUNCTION create_limited_number_policy()
RETURNS void
$BODY$
BEGIN
  CREATE POLICY limited_number ON example_data_set
    USING (limit_reached('example_data_set', 100));
END;
$BODY$
LANGUAGE plpgsql;
```

Figure 11: An example function to create a policy that limits the number of records shared.

B. Complex Sharing Policy Example

An example of a complex policy based on the definition given above is listed in Figure 12. In the example data set, each record is shared based on different conditions described by the simple sharing policies. For records that are associated with multiple simple sharing policies, the conditions according to all those policies must be evaluated to determine if the record should be shared. Policy 1 maps to five records in the database, $p_1 \rightarrow \{r_1, r_4, r_5, r_6, r_8\}$. Further, Record 8 (described by policies p_1 and p_2) can only be shared a limited number of times and for a limited time period. A row level security policy as shown in Figure 13 can be applied in PostgreSQL to enforce this sharing policy.

The `eval_complex_policy` function retrieves the bit string from the Min Mask Sketch associated with the current table (the Min Mask Sketch structure is described

ID	Policy Name	Policy Condition
1	Limited Time Period	USING expression of Figure 3
2	Random Sample	USING expression of Figure 5
3	Biased Sample	USING expression of Figure 7
4	Set of Other Users	USING expression of Figure 9
5	Limited Number of Records	USING expression of Figure 11

(a) The top table is a mapping table for the policy type and the condition to be evaluated for a particular policy. The policy cones teh using expression of the linked function.

Policy ID	Record
1, 3	Record 1
2, 4, 5	Record 2
5	Record 3
1, 3, 4, 5	Record 4
1, 4	Record 5
1	Record 6
2, 4	Record 7
1, 2	Record 8
2, 4	Record 9

(b) The bottom table is an assignment of policies over a data set.

Figure 12: Each table shows an example of a complex sharing policy applied to a data set.

```
CREATE FUNCTION create_complex_policy()
RETURNS void
$BODY$
BEGIN
  CREATE POLICY complex_policy ON example_data_set
    USING (eval_complex_policy('example_data_set'));
END;
$BODY$
LANGUAGE plpgsql;
```

Figure 13: An example function to instantiate a complex policy.

in Section IV). For each active policy, the condition would be retrieved from the mapping table shown in Figure 12. If all the conditions evaluate to `true`, then the function will return `true` and the record will be shared. If any of the conditions evaluate to `false`, the function will return `false` and the record will be restricted.

Work has been done in the database community to develop methods for implementing data sharing policies within Hippocratic database systems [3]. Language constructs have been created to define these fine grained access control policies with minimal complexity [4]. One goal for minimizing the complexity of these policy representations is to reduce the storage overhead on a database management system that implements fine grained access control. In this paper, we introduce a new method for storing policy meta data that aims to further reduce the cost of storage.

III. Use Case

Consider a new mobile application, Health Tracker Pro, that uses a health monitoring device to record a user's fitness data. The purpose of this application is to not only help users monitor their personal health, but also give them the ability to share their personal health data with their doctor. Doctor's

time	heart_rate	blood_sugar	body_temp
2016-02-20 04:05:06	71	95	98.6
2016-02-20 04:05:09	72	96	98.7
2016-02-20 04:05:12	72	94	98.7
⋮			
2016-02-21 11:14:40	115	125	99.3
2016-02-21 11:14:43	115	124	99.5
2016-02-21 11:14:46	116	124	99.6

Figure 14: Example table containing Bob's personal health data recorded by Health Tracker Pro.

would use the Health Tracker Pro Dashboard application to view health data shared by each of their patients.

The primary data recorded by Health Tracker Pro is stored in a single table. This table has the following schema (using PostgreSQL data types):

```
health_data(
    time          timestamp    primary_key
    heart_rate    smallint     not null
    blood_sugar   smallint     not null
    body_temp     real         not null
)
```

Health Tracker Pro uses a sampling rate of 20 times per minute, or more precisely, once every three seconds. An example subset of this data can be seen in Figure 1 for an example user, Bob.

Bob would like to share some of his health data with his doctor. However, he does not want to share all of the data recorded by Health Tracker Pro. Bob only wishes to share his data at certain times during the day. For example, Bob would like to share his heart rate and body temperature data while exercising, and blood sugar data while sleeping and after eating a meal. Additionally, if Bob's heart rate is recorded to be outside of a selected window, he would like his doctor to be notified. At all other times of the day, Bob would like his personal health data to remain private.

The simplest approach to storing these complex privacy policies would be by adding three new attributes to the health_data table as shown below:

```
health_data(
    time          timestamp    pimary_key
    heart_rate    smallint     not null
    blood_sugar   smallint     not null
    body_temp     real         not null
    hr_private    boolean      not null
    bs_private    boolean      not null
    bt_private    boolean      not null
)
```

These attributes are simple Boolean values determining whether the corresponding attribute for a row is private. Each row may represent a unique policy, so the policy data will be stored in the same table alongside the primary health data. In the following sections, we will describe the Min Mask Sketch data structure and implementation that can store these privacy policies without using near as much space. We will then discuss the pros and cons of this data structure when compared to the simple method described here as well as briefly mention another alternative approach to storing these complex sharing policies.

IV. Min Mask sketch

The Min Mask Sketch is a modified version of the Count Min Sketch [2]. The Min Mask Sketch is stored as a two-dimensional array of unsigned integers and uses a collection of hash functions. The purpose of the Min Mask Sketch is to efficiently store policies associated with items in a large data set. When the sketch is first created, all elements in the two-dimensional array are initialized to zero. When a new item's policy is inserted into the sketch, the item is hashed by d different hash functions, where d is the number of rows in the two-dimensional array. These hash functions return a uniformly random value between 0 and w-1, where w is the number of columns in the two-dimensional array. The policy for the given data item is then inserted into a cell contained in each row of the two-dimensional array at the particular index calculated by the corresponding hash function. This process is illustrated in Figure 15.

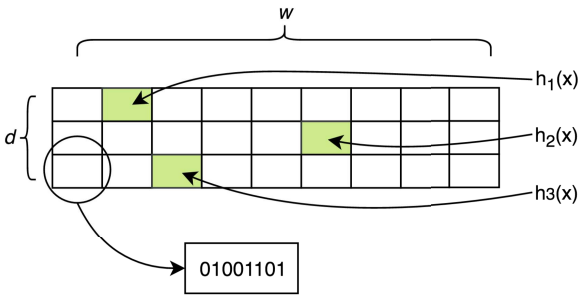

Figure 15: The Min Mask Sketch

At time of insertion, the policy should be in the form of an unsigned integer. We will refer to this unsigned integer as the bitmask for that policy. The bitmask approach is very simple. Each bit position in the bitmask corresponds to a possible condition in the complex sharing policy associated with a data item. If the bit at a particular position is 1, then the condition corresponding to that bit position is *active* for that item and should be applied when the data is shared. To insert the policy value into a cell, a bitwise OR operation is performed with the existing bitmask in the cell and the new bitmask to be inserted. This is done to avoid overwriting existing bitmasks in the sketch when a hash collision occurs or when performing an update to an existing item. Inserting a new item into the Min Mask Sketch is a straightforward process. Updating an existing item presents new problems which will be discussed in Section VI.

Using the running example from Section III, suppose the least significant bit position in the bitmask corresponds to making Bob's body temperature information private, the next highest order bit position corresponds to making Bob's blood sugar information private, and the next highest order bit position corresponds to making Bob's heart rate information private. During exercise, Bob would like to share his heart rate and body temperature information with his doctor. The corresponding sharing policy for each data item recorded during his exercise session would be "010". Meaning that his heart rate and body temperature are freely being shared, but his blood sugar has an active "private" condition associated with it that should be applied during the sharing process. Since the time attribute is the primary key for the health_data table in the example given in Section III, the time value for each data item would be the argument passed into each hash function.

When retrieving a policy bitmask for a given item, the primary key for that item is hashed by all d hash functions to get the indexes that should be checked for each row in the two-dimensional array. The bitmask values at each index are passed into a function that determines the number of 1's, or "active policies", contained in each bitmask and returns the bitmask with the minimum number of active policies as the result. Due to hash collisions, this bitmask is only an estimate for the actual bitmask associated with the given data item. This estimate is bounded by the following equation with probability c:

$$a \leq \hat{a} \leq a + \epsilon \mathcal{M} \tag{1}$$

Where a is the actual policy for a given item, $\hat{a}$ is the estimated policy, and ϵ is the error bound factor. This equation has been formally proven by Cormode and Muthukrishnan in the original paper describing the Count Min Sketch [2].

In Cormode et al. [2] $\mathcal{M}$ represents the sum of all of the true frequencies in the Count Min Sketch. Because the Min Mask Sketch uses bit positions and not frequency values, the equivalent $\mathcal{M}$ is described in Equation 2 as the sum of 2 raised to the Hamming weight of each a_i in the Min Mask Sketch.

$$\mathcal{M} = \sum_{i}^{w*d} 2^{\text{weight}(a_i)} \tag{2}$$

Since this equation is known to be true, the estimate for the policy associated with a given item in the sketch will always be either the correct policy or contain a small amount of extra 1's in the policy. The design of the Min

Mask Sketch was done in such a way to err on the side of caution assuming by default that all data is shared. When the default behavior of the system is to not share data, the errors in estimation due to the probabilistic nature of the Min Mask Sketch may result is sharing data that is not meant to be shared. This may not be desirable behavior for many use cases.

c, the confidence interval for the error bound, and ϵ,the error bound factor, can be chosen at creation time to fit the sketch with the needs of the particular application. The smaller the error bound factor, and the greater the confidence interval, the more space is needed for the Min Mask Sketch to deliver these guarantees. This is because the width and depth of the two-dimensional array used for the Min Mask Sketch are determined based on these two parameters:

$$w = \lceil \frac{e}{\epsilon} \rceil \tag{3}$$

$$d = \ln(\frac{1}{1-c}) \tag{4}$$

Cormode and Muthukrishnan did extensive theoretical analysis, proving that when the sketch is sized in this manner and d hash functions are used, (1) holds true. The size of the resulting Min Mask Sketch does not grow as a function of the data set, it is completely fixed based on the tuning of c and ϵ. However, ϵ should be chosen with the number of insertions kept in mind. Since the upper bound for the estimation error is a result of multiplying the number of insertions by the error bound factor ϵ, a value for ϵ can be chosen to tune this upper bound to a precise point based on the estimated number of insertions that will occur in the Min Mask Sketch. If ϵ is chosen to be too large compared to the number of insertions, the upper bound for the error in estimating a policy will grow significantly and the Min Mask Sketch will no longer be useful.

V. Implementation

Our implementation of the Min Mask Sketch was done by creating an extension for PostgreSQL version 9.6 [1]. The extension was written in C and contains four major components:

1) Definition of the Min Mask Sketch data type.
2) Functions to create a new Min Mask Sketch object.
3) Functions to add an item into the Min Mask Sketch.
4) Functions to retrieve the bitmask for a given item in the Min Mask Sketch.

The first component of the extension was implemented by creating a simple C structure containing three fields: two integer variables to hold the sketch depth and sketch width, and an array of integers to represent the sketch itself. This C structure is then mapped to a PostgreSQL data type called "mms" that can be attributed to a column in a `CREATE TABLE` statement. For example, to create a table containing a column with the Min Mask Sketch data type, the following SQL can be executed:

```
CREATE TABLE example (
    my_sketch mms
);
```

Creating this table does not automatically instantiate a new Min Mask Sketch object. This is where the second component of the extension is required. In order to instantiate a new Min Mask Sketch object, we created a user-facing function called "mms" that accepts two parameters. These parameters are floating point numbers corresponding to the error bound and confidence interval for the sketch. These are optional parameters with default values of 0.001 and 0.99 respectively. The error bound and confidence interval are then used to determine the sketch depth and sketch width. This process was discussed in detail in Section IV. The required amount of memory for the sketch array is then allocated and each value in the sketch is initialized to 0. The new Min Mask Sketch object is then returned. In order to insert a new Min Mask Sketch object into the example table created above, the following SQL code can be executed:

```
INSERT INTO example VALUES(mms());
```

The third component of the extension handles adding new items into the sketch, and was implemented by creating a user-facing function called "mms_add". This function takes three parameters: the sketch to which the new item should be added, the new item itself, and a bitmask to identify the policy that should be applied to the item when being shared (as described in Section IV). The new item is first hashed, using MurmurHash3, to d different locations in the sketch, where d corresponds to the sketch depth calculated at creation time. The values computed by the hash functions correspond to indexes in the sketch array. A bitwise OR operation is then performed between the existing value in the sketch at each index and the new bitmask value given as the third function argument. This successfully adds or updates the item accordingly. An example SQL statement to add a new item into a Min Mask Sketch object is given below:

```
UPDATE example SET my_sketch =
    mms_add(my_sketch, "abc"::text, 6);
```

Note in the example above that the new item is of type "text", but any data type is supported, and the integer "6" corresponds to the binary representation "110", meaning two conditions are active for that item. It is possible to update an item to use a new policy, however, the update must only result in bits changing from 0 to 1, not the reverse. In other words, new conditions for a specific row can be set to active but new existing active conditions cannot be deactivated. This limitation is discussed in more detail in Section VI.

The fourth component of the extension was implemented by creating a user-facing function to retrieve the bitmask associated with a given item in the data set. This function

[1]The implementation is available on GitHub at https://github.com/oudalab/mms

is called “mms_get_mask” and takes two parameters. The first parameter is the sketch in which the item is stored, and the second parameter is the item in question. This function hashes the given item to obtain the d different hash values corresponding to the indexes in the sketch that must be checked. The bitmasks at each index are retrieved from the sketch and the minimum mask value is calculated (as described in Section IV). This minimum mask value is then returned to the user as the policy for that data item. An example SQL statement to retrieve the bitmask value associated with an item in a Min Mask Sketch is as follows:

```
SELECT mms_get_mask(my_sketch, "abc"::text)
    FROM example;
```

This query returns the bitmask value associated with the item “abc” which can then be used to determine the policy that should be applied to the row identified by “abc”. If the item does not exist in the data set, this function will return 0.

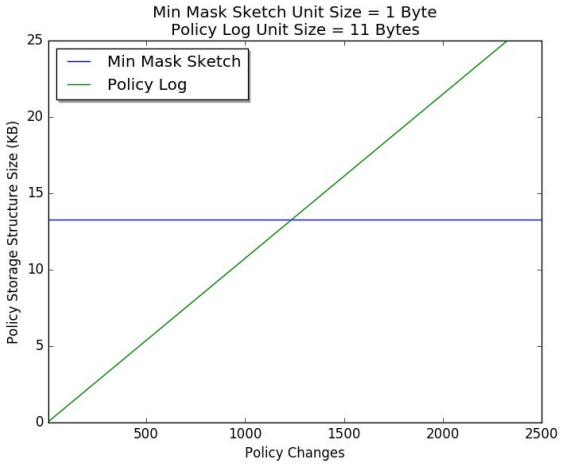

Figure 16: Comparing the space efficiency of the Min Mask Sketch vs the log-based storage approach.

VI. Analysis

After analyzing this approach to store complex sharing policies efficiently, several issues have arisen that will be discussed in this section. The first issue is the limitation that the Min Mask Sketch has in regards to handling updates and deletions. The Min Mask Sketch approach only succeeds in handling updates that result in changing bits from a 0 to a 1 in the bitmask and not the reverse. The sketch cannot handle deletions at all. This is due to the fact that when retrieving a policy for a given data item, the policy corresponding to the minimum number of 1’s is chosen. Therefore, if a policy was updated from “111” to “001”, and one of the indexes calculated by the hash functions collided with a second item in the sketch, this could potentially ruin the accuracy of that second item. If the second item involved in the hash collision had a policy of “101”, one of its d bitmasks would be changed to “001” by the update to the first policy, resulting in a new minimum bitmask for the second item that is inaccurate. Deleting an item is a problem for the exact same reason, because it results in bitmasks going from a higher number of 1’s to a lower number of 1’s. The Count Min Sketch and other modifications to the Bloom Filter such as the Counting Bloom Filter are able to handle updates and deletions using increments and decrements [5]. The Min Mask Sketch, however, does not inherit this functionality because performing a bitwise logical OR operation between two integers is not the same as incrementing and thus information can be lost when a hash collision occurs. One approach to solving this problem would be by choosing the average bitmask value instead of the minimum, however this would result in a looser error bound and thus more inaccuracies.

The second issue with this approach to storing complex policies is the fact that the simplest way of storing the policy does not add a large amount of overhead, so the introduction of inaccuracies when applying sharing policies may not be worth it. When considering the example from Section III, one entry in the health_data table can be stored in 16 bytes (excluding the three additional columns added to store policies), while the policy data can be stored in 3 bytes using the simple approach of adding three extra columns. With an overly generous assumption that the Min Mask Sketch approach could store all of the policies in a negligible amount of space, this would result in an 18.75% space efficiency increase. This efficiency is reasonable, but it should be noted that this percentage value is an upper bound to the space efficiency increase for this example and would only shrink as more health data was recorded by the Health Tracker Pro app. Since the Min Mask Sketch approach brings potential inaccuracies in the estimation of policies, it should result in a space efficiency increase large enough to warrant those inaccuracies. If a more complex policy were used that would require even more data to represent it than the primary data itself, the Min Mask Sketch approach to storing these policies would become much more feasible, but practical examples that involve such large complex policies are scarce.

The final issue that will be analyzed here is related to the frequency of policy changes within a large data set and their role in the feasibility of the Min Mask Sketch approach. Consider the running example from Section III. Based on Bob’s wishes, the policies associated with each row would only change a few times per day. This means that most rows in the health_data table will contain the same complex policy. When there are very few policy changes, an alternative method for storing these policies could be used that is based on storing a policy for a range of items in the table. We will call this approach the log-based approach. For example, if Bob’s sharing policy only changes 6 times

per day, 6 entries in a log table could be inserted, where each entry contained a timestamp, and a Boolean value for each condition. The time between each entry in the log table would be the range for those policies to be applied. When determining the policy for a given data item, the log table could be referenced and the range of times given by the different timestamps would determine which policy should be applied to that item. This approach is much more efficient than other approaches when the frequency of policy changes is low.

The size of the log table grows as a function of the number of policy changes within a data set. Figure 16 shows the storage space used for the Min Mask Sketch compared to the space used for the log-based approach for a variety of policy changes based on the running example from Section III. In this graph, the default error bound factor and confidence interval were used (0.001 and 0.99 respectively). As one can clearly see, the log-based approach outperforms the Min Mask Sketch approach for the Health Tracker Pro example until roughly 1250 policy changes occur in the data set. Also, the Min Mask Sketch approach introduces inaccuracies due to the probabilistic behavior of the data structure, therefore, the Min Mask Sketch storage approach would need to significantly outperform the log-based approach for it to be a practical choice.

VII. Conclusion

Complex data sharing policies are becoming increasingly common as more applications are recording data and sharing it across a large network of devices and people. We have presented the Min Mask Sketch approach to efficiently store these policies. We have also described our implementation for the Min Mask Sketch within the PostgreSQL 9.6 database management system. After a detailed analysis of some of the key factors involved in storing complex sharing policies, we have seen that there are several issues with this probabilistic approach to storing complex sharing policies. We have discussed these issues in detail in order to understand the fundamental questions that need to be answered when developing solutions for storing complex sharing policies. Some of the problems discussed in the analysis section can be solved through future work and design changes, while other problems require a better understanding of how data might be shared in the future in order to solve.

VIII. Acknowledgements

We would like to thank Sai Ram Sunkara for his help during the initial implementation.

References

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[2] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[3] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, "Limiting disclosure in hippocratic databases," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 108–119.

[4] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi, "Extending relational database systems to automatically enforce privacy policies," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*. IEEE, 2005, pp. 1013–1022.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.