Making Postgres Bloom

Usman Masood on July 30, 2015

Whenever you think about low-latency stream computation, the key thing is figuring out how to summarize raw data, which might be too voluminous to compute on at low latency, while still maintaining the ability to answer relevant queries. For simple additive queries like counts and averages, it's pretty straightforward to condense the raw data down to a few numbers. But for more complex queries like calculating percentiles, heavy hitters, etc., summarizing the raw data becomes a little more tricky. Your fundamental trade-off in such cases is to either become a resource hog (read: lots of money) or lose a bit of precision by using a compact probabilistic summary data structure.

In this post, I'll show you how probabilistic data structures can be used to approximate answers quickly, how custom data types can be implemented natively in Postgres, and an overview of how PipelineDB adds extremely fast stream computation capabilities to Postgres.

Let's go.

An example: Knowing Who Did What on Your Site

Suppose you want to know whether a user has done a certain action on your website or not. The resource hog way would be to record an event every time a user takes that action, say in a `user_actions` table and then scan through the entire table every time you want to answer that query.

```
SELECT <user_id> IN (SELECT DISTINCT user_id FROM user_actions);
```

`<user_id>` *is an ID of some user, for example 'usmanm'.*

Okay, sure, this might be overkill. You really only need store the user IDs, so you can save a bit of space there. In most real-world cases, the number of actions is going to be much larger than the number of unique users, which means the `DISTINCT` query can be painfully slow. Maybe you could create a `MATERIALIZED VIEW` and query that instead, but that comes with its own can of worms.

Let's not over-engineer though. The problem here is classically described as the set membership problem. We have a set of users that performed some action, and we want to know whether a particular user belongs to that set or not. You cannot get space usage better than linear in the number of distinct users if 100% accuracy is expected, but under a streaming setting that can be prohibitively large.

Meet Bloom Filters

But what if you only want to be really accurate, rather than getting a precise result? Bloom filters are a space-efficient probabilistic data structure used to answer set membership queries. They have a tunable non-negative false positive probability, but never report a false negative. In the case of our users example, this means we'll never miss a user that has done the action, but sometimes might get a yes for someone who didn't.

Here's a quick overview of Bloom filters: you take a bit array of size `m`, and `k` hash function which return an integer in the range `[0, m)`. To add an element to the filter, you hash it one by one using each of the `k` hash functions to get `k` locations, and then set the bit at those locations to `1`. To check whether an item is in the filter or not, you check each of the `k` locations returned by the hash functions and return true only if all of them were set to `1`.

If you want more details, here's a super layman explanation, and this is the original paper published by Mr. Bloom himself. Pick your poison.

CREATE TYPE

So yeah, Bloom filters are cool, But weren't we talking about Postgres? Turns out Postgres has a pretty slick type system that lets you define custom types. Time to create a Bloom filter type!

Thinking about what we need to store in the type, we want the length of the bit array `m`, the number of hash functions `k`, and obviously the bit array `bits` itself.

```
CREATE TYPE dumbloom AS (
  m    integer,
  k    integer,
  -- Our bit array is actually an array of integers
  bits integer[]
);
```

Next, we need a constructor to create an empty Bloom filter. We'll also give it two optional arguments, `p` and `n`. `p` is the false probability we can tolerate and `n` is the number of *distinct* elements we expect to insert into the filter. Yes, you need to know the value of `n` a priori which kind of sucks, but there are funkier Bloom filters like Scalable Bloom filters and Stable Bloom filters which do away with that necessity at a cost (increased space over time or non-zero false negative probability respectively).

You're going to see a lot of PL/pgSQL now, but you can use other languages as well!

```
CREATE FUNCTION dumbloom_empty (
  -- 2% false probability
  p float8 DEFAULT 0.02,
  -- 100k expected uniques
  n integer DEFAULT 100000
) RETURNS dumbloom AS
$$
```

```
DECLARE
  m    integer;
  k    integer;
  i    integer;
  bits integer[];
BEGIN
  -- Getting m and k from n and p is some math sorcery
  -- See: https://en.wikipedia.org/wiki/Bloom_filter#Optimal_number_of_hash_functions
  m := abs(ceil(n * ln(p) / (ln(2) ^ 2)))::integer;
  k := round(ln(2) * m / n)::integer;
  bits := NULL;

  -- Initialize all bits to 0
  FOR i in 1 .. ceil(m / 32.0) LOOP
    bits := array_append(bits, 0);
  END LOOP;

  RETURN (m, k, bits)::dumbloom;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

We'll create a utility function to fingerprint each incoming item. For the purpose of this post, our Bloom filter implementation will only accept items of `text` type (or types that can be coerced into a `text` type). The fingerprint in our case is an array of length `k` where each element in the array is the output of a different hash function.

```
CREATE FUNCTION dumbloom_fingerprint (
  b    dumbloom,
  item text
) RETURNS integer[] AS
$$
DECLARE
  h1     bigint;
  h2     bigint;
  i      integer;
  fingerprint integer[];
BEGIN
  h1 := abs(hashtext(upper(item)));
  -- If lower(item) and upper(item) are the same, h1 and h2 will be identical too,
  -- let's add some random salt
  h2 := abs(hashtext(lower(item) || 'yo dawg!'));
  finger := NULL;

  FOR i IN 1 .. b.k LOOP
    -- This combinatorial approach works just as well as using k independent
    -- hash functions, but is obviously much faster
    -- See: http://www.eecs.harvard.edu/~kirsch/pubs/bbbf/esa06.pdf
    fingerprint := array_append(fingerprint, ((h1 + i * h2) % b.m)::integer);
  END LOOP;

  RETURN fingerprint;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Writing the add and contains routines is pretty trivial now.

```
CREATE FUNCTION dumbloom_add (
  b    dumbloom,
  item text,
) RETURNS dumbloom AS
$$
DECLARE
  i    integer;
  idx  integer;
BEGIN
```

```
  IF b IS NULL THEN
    b := dumbloom_empty();
  END IF;

  FOREACH i IN ARRAY dumbloom_fingerprint(b, item) LOOP
    -- Postgres uses 1-indexing, hence the + 1 here
    idx := i / 32 + 1;
    b.bits[idx] := b.bits[idx] | (1 << (i % 32));
  END LOOP;

  RETURN b;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

CREATE FUNCTION dumbloom_contains (
  b    dumbloom,
  item text
) RETURNS boolean AS
$$
DECLARE
  i   integer;
  idx integer;
BEGIN
  IF b IS NULL THEN
    RETURN FALSE;
  END IF;

  FOREACH i IN ARRAY dumbloom_fingerprint(b, item) LOOP
    idx := i / 32 + 1;
    IF NOT (b.bits[idx] & (1 << (i % 32)))::boolean THEN
      RETURN FALSE;
    END IF;
  END LOOP;

  RETURN TRUE;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Boom! We're done. A fully functional native implementation of Bloom filters in ~100 lines of code--Postgres, I love you. Let's give this puppy a try now.

```
CREATE TABLE t (
  users dumbloom
);

INSERT INTO t VALUES (dumbloom_empty());

UPDATE t SET users = dumbloom_add(users, 'usmanm');
UPDATE t SET users = dumbloom_add(users, 'billyg');
UPDATE t SET users = dumbloom_add(users, 'pipeline');

-- This first three will return true
SELECT dumbloom_contains(users, 'usmanm') FROM t;
SELECT dumbloom_contains(users, 'billyg') FROM t;
SELECT dumbloom_contains(users, 'pipeline') FROM t;
-- This will return false because we never added 'unknown' to the Bloom filter
SELECT dumbloom_contains(users, 'unknown') FROM t;
```

And it works! You should see `true` for the first three queries but `false` for the last one because we never added 'unknown' to our Bloom filter.

Some of you, who have ventured deep into the bowels of databases, will probably point out that doing something like this in a real setup is committing concurrency suicide. All updates to the same row will essentially be executed serially which is no bueno if you're trying to build a performant data pipeline.

Enter PipelineDB

So how do we avoid this? The core idea is to process a bunch of incoming tuples in batches and commit a batch to disk once, rather than hitting disk for every single incoming tuple. Batches can also be processed in parallel without any lock contention if only a single process is responsible for committing to disk. This is what PipelineDB excels at! Our continuous view abstraction handles this batching transparently for all supported aggregates.

The key to all of this is having summary states which are "mergeable". For example, for something like `count`, the merge operation is `sum`. You process a batch to get the count of tuples in it, and then simply sum it with the value of the on-disk tuple and commit. PipelineDB's continuous query executor has two types of processes: workers and combiners. Workers are responsible for reading tuples that are inserted into a stream and generating summaries for the micro-batches. Once these summaries are generated, they're passed on to a combiner process which merges them with on-disk values and commits the updated summary states to disk.

There are two key properties that a merge operator must have: commutativity and associativity. You can think of these summary structures as being a poor man's CRDT, the poorness coming from a lack of idempotency. As long as you can create a summary structure which can be merged using a commutative and associative operator, PipelineDB can handle it.

So back to Bloom filters. How do we merge them? Well, Bloom filters are essentially representations of sets, so we just need to figure out how to union two Bloom filters. Turns out it's fairly easy: you just take a bit-wise OR of the bit set of two Bloom filters to get a Bloom filter that represents the union of the two sets.

```
CREATE FUNCTION dumbloom_union (
  b1 dumbloom,
  b2 dumbloom
) RETURNS dumbloom AS
$$
DECLARE
  i    integer;
  bits integer[];
BEGIN
  IF b1 IS NULL THEN
    RETURN b2;
  END IF;

  IF b2 IS NULL THEN
    RETURN b1;
  END IF;

  IF NOT (b1.m = b2.m AND b1.k = b2.k) THEN
    RAISE EXCEPTION 'bloom filter mismatch!';
  END IF;

  bits := NULL;

  FOR i IN 1 .. ceil(b1.m / 32.0) LOOP
    -- Bitwise OR the two bitsets
    bits := array_append(bits, b1.bits[i] | b2.bits[i]);
  END LOOP;

  RETURN (b1.m, b1.k, bits)::dumbloom;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
```

Now we're ready to create a PipelineDB supported aggregate for Bloom filters. We piggyback on Postgres' `CREATE AGGREGATE` command, except that we need to pass a `combinefunc` parameter to the command as well.

```
CREATE AGGREGATE dumbloom_agg (text) (
  -- the transition state type
  stype = dumbloom,
  -- function to *advance* the transition state, like adding 1 for count
  sfunc = dumbloom_add,
  -- function to merge the transition state, like sum for count
  combinefunc = dumbloom_union
);
```

The worker processes will add incoming values using `dumbloom_agg` to create a single Bloom filter for a micro-batch, and the combiner will continuously merge all the incoming Bloom filters from workers with the on-disk one using `dumbloom_union`.

Real or Vapor?

Let's put this to the test. We're going to create a table with a`dumbloom` field and update it repeatedly by adding some random text. To compare with, we're going to create a continuous view that does a `dumbloom_agg` over a `text` field.

```
CREATE TABLE test (
  b dumbloom
);

CREATE CONTINUOUS VIEW test_cv AS
  SELECT dumbloom_agg(x::text) FROM stream;
```

We'll test using 100k data points. The `table_updates` function is just a simple loop that updates the `b` field of table `test`.

```
CREATE FUNCTION table_updates(
  n integer
) RETURNS void AS
$$
DECLARE
  i integer;
BEGIN
  FOR i IN 1 .. n LOOP
    UPDATE test
      SET b = dumbloom_add(b, left(md5(random()::text), 3));
  END LOOP;
END
$$
LANGUAGE 'plpgsql';

INSERT INTO t VALUES (dumbloom_empty());
INSERT 0 1

SELECT table_updates(100000);
 table_updates
---------------

(1 row)

Time: 343721.515 ms

INSERT INTO stream (x)
  (SELECT left(md5(random()::text), 3) AS x FROM generate_series(0, 100000));
INSERT 0 100001
Time: 19691.468 ms
```

6 minutes vs 20 seconds? Clearly, continuous views are significantly faster compared to the naive way of repeatedly updating columns. But 20s for a 100k `INSERT`s is still pretty slow... Let's see how this compares to our native `bloom_agg`, which is a Bloom filter implementation in C baked into PipelineDB.

```
-- We provide the value for p and n because the defaults for our native implementation are
-- 0.02 and 16384, so it would be an unfair comparison
CREATE CONTINUOUS VIEW test_native_cv AS
  SELECT bloom_agg(x::text, 0.02, 100000) FROM stream_native;

INSERT INTO stream_native (x)
  (SELECT left(md5(random()::text), 3) AS x FROM generate_series(0, 100000));
INSERT 0 100001
Time: 466.216 ms
```

Less than half a second, that's more like it!

I agree that each of these queries can be optimized further. My only goal here is to motivate the performance and resource utilization benefits of continuous views.

What's the Catch?

Okay, so I may have cheated a bit. Continuous views are much faster than table `UPDATE`s but `INSERT`s into streams don't support any transaction semantics (at least not for now). There's two insertion modes: synchronous and asynchronous. Setting the `synchronous_stream_insert` configuration parameter toggles this behavior. Asynchronous mode is much faster, because it inserts the tuple into an in-memory buffer and returns immediately without waiting for it to be consumed. The synchronous mode only returns once the state changes resulting from the inserted tuple have been committed. However, if inserts are happening concurrently then state changes from insertions happening in different *transactions* could be committed in a single batch.

Show Me Moar

Enough about Bloom filters, let's talk about what else PipelineDB has to offer. Other than the standard Postgres aggregates which we (almost) fully support, we've also implemented a few probabilistic data structures that are used to answer common queries you might have.

- HyperLogLogs can help you answer set cardinality questions, like the number of unique visitors.
- Count-min Sketches let you estimate frequency of elements seen.
- T-Digests can be used to estimate quantiles or the value at a given quantile, like what is the 99th percentile latency.
- Bloom Filters... um, I hope you were paying attention.

We've received a few requests about adding something that lets you estimate top-K frequent elements. Well, we've listened and it's primed for the next release.

Final Thoughts

Stream computation is increasingly becoming a standard part of everyone's data infrastructure stack. While previously you had to glue heavy weight systems together, PipelineDB offers a simple way to get started. The goal is to keep on adding more online algorithms to our repository so you can leverage PipelineDB for much more advanced analytical work than what is possible with vanilla Postgres. If there's something you need in particular, give us a shout out!

If any of this sounds interesting to do for a living, we're hiring!