# Faster exact match searches with Bloom filters in PostgreSQL

Jacob Warwick
University of Washington
Seattle WA, USA
jacobw42@uw.edu

## Introduction

Across distributed and non-distributed database systems, nearly every query plan begins with a scan operation. And regardless of the specific system, query, or data storage format, scans are expensive; the physical limits of hardware impose non-negotiable constraints. To overcome this, databases create indices, data structures that allow the database engine to locate and retrieve only the relevant data from the disk. Commonly, these indices are implemented as binary trees or hash tables. However, in cases where data are extremely large and diverse, building and maintaining an index can be expensive.

Bloom filters are a statistical data structure that allow for efficient existence queries of sets with constant complexity. They also allow for the addition, but not deletion, of new data after the filter has been constructed. Querying a bloom filter may produce a false positive; the false positive rate is a function of the size of the bloom filter and the number of hash functions used.

This project was motivated by a need to search for exact matches across a large set of highly unique strings. Unlike a web search problem, searching for exact matches meant that Bloom filters could theoretically be used to remove groups of strings from consideration, effectively reducing the scope of the search space before running a traditional scan on the smaller space.

## Evaluated Systems

The functionality to create or query Bloom filters does not currently exist in either traditional or distributed database systems. The natural choice of programming language for a custom implementation was C, for its fluency with low-level memory operations, and its speed. Choosing C necessitated that I work within the PostgreSQL database, which in my research was one of two database systems supporting user-defined functions in C (the other was Apache Impala).

PostgreSQL 12 is the most recent stable release, and it includes support for parallel scan query plans, where parallel threads are used on large scans within the same query. I hoped that by implementing my bloom filters in C and querying them through Postgres, I could take advantage of Postgres' parallelism and storage manager.

## Problem Statement and Method

Implementing a custom index for PostgreSQL was beyond my capabilities and scope, so instead I implemented a C library and a C++ program to create a bloom filter for each biological sample in my dataset, and load them into the PostgreSQL database as byte arrays. I then reused my library to build a UDF to query for search sequences within those byte arrays. Bloom filter creation and database ingestion took about 1 hour for 1.6 billion sequences across 22,827 samples, with each bloom filter constructed for a 1e-12 false positive rate. The resulting data structures were approximately 11GB on disk.

I conducted my tests on a large virtual machine, with 32 virtual cores and 250GB of RAM. I configured my Postgres server to take advantage of all available resources.

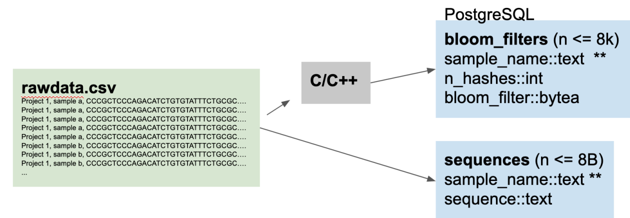


**Figure 1: Ingestion process and resulting schema**

I wrote a SQL query that uses the Bloom filters to narrow the search space to just samples with matching sequences, and then performs an exhaustive search on that subset of samples. In the next section, I compare the performance of that query to that of a simple parallelized hash-join.

While PostgreSQL is not a distributed system, the problem is trivially parallelizable, so my intent is to explore the effectiveness of this approach, with the understanding that it could be applied to parallel systems too. I ran experiments on a virtual machine with 32 virtual processors and 256 GB of RAM, and modified my server config as follows, to take advantage of the available resources (once my test query was established, I experimented with several versions of these hyperparameters):

```
shared_buffers = 30GB # from the default 128MB
temp_buffers = 16GB                    # min 800kB
work_mem = 20GB                        # min 64kB
effective_io_concurrency = 8
max_worker_processes = 32
max_parallel_maintenance_workers = 32
max_parallel_workers_per_gather = 32
```

```
max_parallel_workers = 32
enable_partitionwise_join = on
enable_partitionwise_aggregate = on
parallel_setup_cost = 1.0
min_parallel_table_scan_size = 16kB
min_parallel_index_scan_size = 16kB
```

My initial attempts resulted in vastly degraded performance due to the schema of the “sequences” table. While Postgres was able to scan the bloom filters relatively quickly in my initial tests, it wasn’t effectively using the results to reduce the amount of table scanning on the sequences table. To address this, I used a CLUSTER index on sample_name, which was also a btree index. This re-sorted the data on disk by sample_name, which gave the Postgres query planner the ability to scan only the blocks containing samples relevant to the query.

Even when the sequences table was sorted, it took several tries to construct a query where the query planner took advantage of the reduction in sample space. Compare the following queries and their corresponding query-plans (tcrs50 is a table of search terms, tcr_bf is the bloom filter table, and test_tcrs is the equivalent of the sequences table previous described):

```
WITH matching_samples AS (
    SELECT tcr_bf.sample_name, tcrs.nucleotide
    FROM tcr_bf, tcrs50 as tcrs
   WHERE bf_contains(nucleotide, bloom, bf_hashes)
)
SELECT test_tcrs.nucleotide, count(*) as matches
FROM test_tcrs, matching_samples
WHERE test_tcrs.nucleotide = matching_samples.nucleotide
AND test_tcrs.sample_name = matching_samples.sample_name
GROUP BY 1;
```

**Figure 2: Naïve search query**

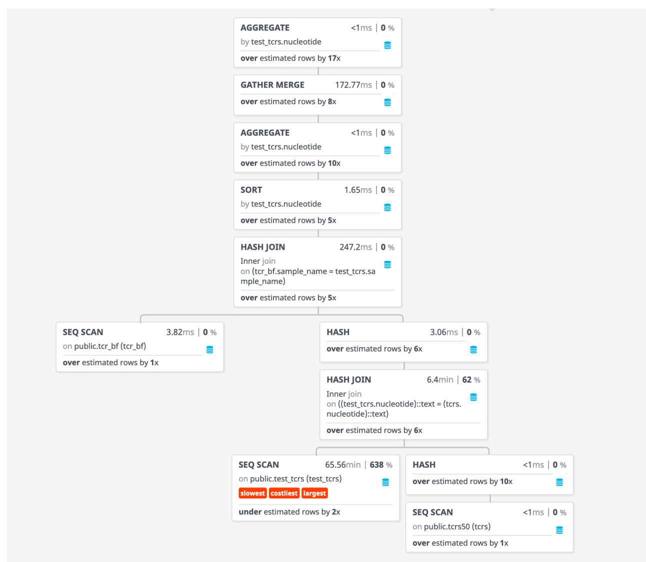


**Figure 3: Naïve search query plan visualized**

```
WITH matching_samples AS (
    SELECT tcr_bf.sample_name, tcrs.nucleotide
    FROM tcr_bf, tcrs50 as tcrs
   WHERE bf_contains(nucleotide, bloom, bf_hashes)
)
SELECT test_tcrs.nucleotide, count(*) as matches
FROM test_tcrs, matching_samples
WHERE test_tcrs.sample_name IN (
    select distinct sample_name from matching_samples
)
AND test_tcrs.nucleotide = matching_samples.nucleotide
AND test_tcrs.sample_name = matching_samples.sample_name
GROUP BY 1;
```

**Figure 4: Improved search query taking advantage of bloom filter reduction**

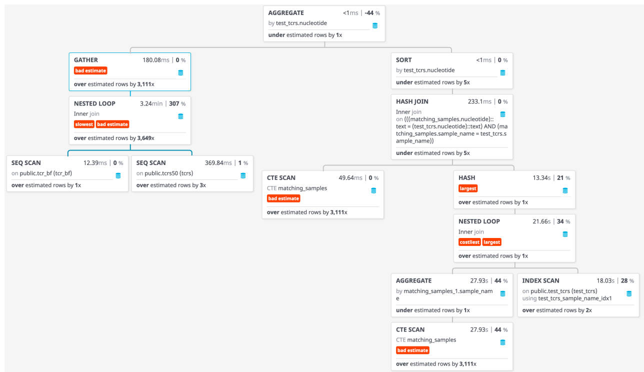


**Figure 5: Improved search query plan visualized**

Postgres’ query planner has two ways to address “WITH” statements: either the named reference is substituted into the query transparently, as a standard subquery, or in cases where the named query is used in a more complicated way or contains side effects, the WITH clause is treated as a common table expression (CTE). CTEs are evaluated and materialized into memory before the rest of the query is run. The critical difference between the query in Figures 2 and 4 is the additional constraint on the final join:

```
WHERE test_tcrs.sample_name IN (
    select distinct sample_name from matching_samples
)
```

Without it, the query is simple enough that the WITH clause is translated into a subquery, and Postgres’ query planner assumes that a hash join between the full sequences table and the search terms is the fastest logical plan. When the extra constraint is applied, Postgres has to materialize the result of the WITH statement before it can begin work on the rest of the query. This materialization allows Postgres to dramatically cut down the amount of scanning on the sequences table, but at the cost of materializing the common table expression.

The goal of this experiment was to measure speedup compared to Postgres’ preferred query plan for this situation, a parallelized hash join. To accomplish this, I ran three versions of the search query against the same test set (of size 16B sequences over 21,755 samples): once using a naïve hash join, one using the bloom filter approach described above, and once just materializing the common

table expression. All durations represented a “warm cache” on the bloom filters table.

The initial test sets consisted of 10, 50, 100, 500, and 1000 sequences that were randomly sampled (uniformly) from the sequences table. This approach produced test sets that actually violated my motivating assumption: that the diversity of my data was high enough, and contamination rare enough, that any set of previously unseen search terms would have very little overlap within the searchable universe. By testing my filter this way, I was effectively testing its performance as the number of matches (or “contaminants”) increased. Detecting 1000 or even 100 matches spread uniformly across the searched universe should be a rare occurrence for which I am not attempting to optimize, and an extremely concerning result in a contamination scenario. By reporting results from this test set, I was able to explore the effect of varying “hit rates” on the filter. In the next section, I refer to matches from this test set as “hits” to clarify that 100% of the test set exists in the search space.

To better understand the scaling properties of my process, I then created more test sets of sizes 100, 1000, 10K, 100K, 1M, and 10M sequences, where each test set contained only 10 sequences in the search space, and the rest of the sequences were randomly generated strings. The matching nucleotides were randomly distributed throughout the test sets.

## Results

In the following chart, the blue bars show the query duration for the hash join, the red bars show the equivalent operation using bloom filters, and the gold bars show just the CTE portion of the bloom filter search query, to aid in the subsequent analysis.

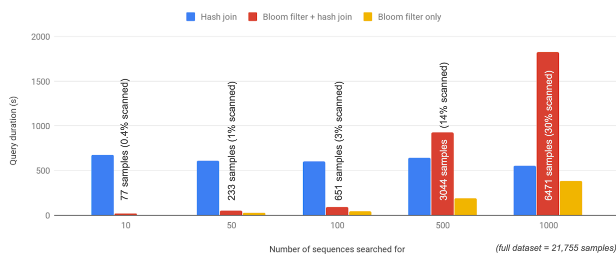


**Figure 6: Query duration with various numbers of query sequences (initial test set)**

Unsurprisingly, Postgres’ parallelized hash join against the full sequences table took a relatively consistent amount of time: each query took about 5 minutes, but that number did not change significantly as the number of search sequences increased. I hypothesize this is because I only tested up to 1000 matching sequences - at a much higher number of search sequences, I would expect hash join to show somewhat degraded performance.

At only 10 matching sequences, the bloom filter query was able to limit the subsequent table scan to 77 samples, or 0.4% of the sequences table, which allowed the entire query to take only 27 seconds compared to the approximately 600 seconds required by hash join. However, as the number of matching sequences increased, so did the time required to scan the bloom filters and materialize the respective matches before the rest of the search could begin. At 500 hits, the bloom filter query exceeded the amount of time taken by the hash join, and at 1000 hits it took almost three times as long as the hash join. Examining the breakdown of CTE-vs-join in the bloom filter query, we can see that just querying and returning 1000 hits from the bloom filters took about 400 seconds. However, it’s hard to tell if that latency was caused by the added overhead of searching for the additional sequences in the bloom filters, or by materializing the significantly longer CTE result.

To figure out the answer, and to gain a better understanding of the scaling characteristics of this approach, examine the results from the same test on the second test set. While I had initially hoped that most of the query time was being spent on tasks other than the bloom filter search, the first test of 10 hits among 100 search terms took 50.95 seconds, and 10 hits out of 1000 search terms took 414.998 seconds. At this point I stopped the test, because it was clear that the search query was scaling along with the number of search terms, not the number of hits, as observed in the previous part of the experiment.

## Conclusion

While the prefiltering approach demonstrated in this paper showed promise with a small number of search terms, it ultimately yielded diminishing returns as the number of search terms increased. In order for this approach to work, it would need to be applied in huge search spaces, with relatively few search terms, and likely in a system with a native integration of the feature.

The computational burden imposed by bloom filter search could likely have been reduced had I chosen to pursue a less extreme false positive rate, or increased the effective IO parallelism parameter of the PostgreSQL server. However, as the intent of this paper was to examine the feasibility of this approach, I felt that further tuning was unnecessary. I have come to feel that the combined prefilter approach - which I originally hoped would be an interesting way to speed up a search query - is only effective in a very narrow window of edge case situations. As such, it is probably better left to a custom-built implementation outside a database system.

## References

Kirsch, Adam, and Michael Mitzenmacher. 2007. "Less Hashing, Same Performance:Building a Better Bloom Filter." *Random Structures and Algorithms* 187-218.

Schmatz, Michael. 2016. *How to write a Bloom filter in C++*. 04 19. Accessed 12 09, 2019. https://blog.michaelschmatz.com/2016/04/11/how-to-write-a-bloom-filter-cpp/.

The PostgreSQL Global Development Group. n.d. *37.10. C-Language Functions* . Accessed 12 09, 2019. https://www.postgresql.org/docs/current/xfunc-c.html.

—. 2019. *11.2 Index Types.* 12 09. www.postgresql.org.
—. n.d. *CLUSTER.* Accessed 12 06, 2019. https://www.postgresql.org/docs/9.1/sql-cluster.html.