Bloom Filters

Christian Antognini
Trivadis AG
Zürich, Switzerland

Oracle Database uses bloom filters in various situations. Unfortunately, no information about their usage is available in Oracle documentation. The aim of this paper is to explain not only what bloom filters are, but also, and foremost, to describe how Oracle Database makes use of them.

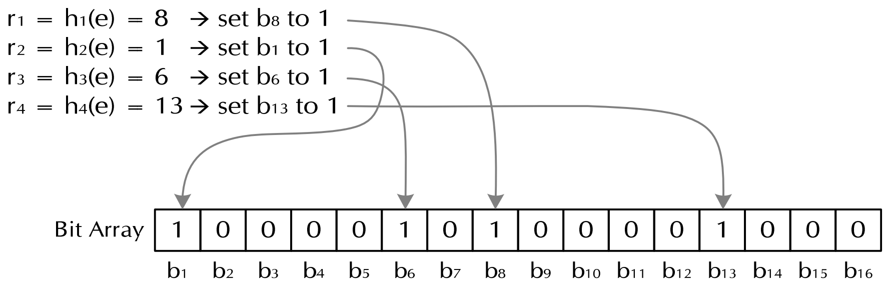
What is a bloom filter?

A bloom filter is not something new or specific to Oracle Database. In fact, it was first developed in 1970 by Burton H. Bloom [1] long before Oracle existed. That being the case, let me explain in general terms what a bloom filter is and for what it can be used for.

A bloom filter is a data structure used to support membership queries. Simply put, a bloom filter is used to test whether an element is a member of a given set or not. Its main properties are:

- The amount of space needed to store the bloom filter is small compared to the amount of data belonging to the set being tested.
- The time needed to check whether an element is a member of a given set is independent of the number of elements contained in the set.
- False negatives are not possible.
- False positives are possible, but their frequency can be controlled. In practice, it is a tradeoff between space/time efficiency and the false positive frequency.

A bloom filter is based on an array of *m* bits (b_1, b_2, …, b_m) that are initially set to 0. To understand how a bloom filter works, it is essential to describe how these bits are set and checked. For this purpose, *k* independent hash functions (h_1, h_2, …, h_k), each returning a value between *1* and *m*, are used. In order to "store" a given element into the bit array, each hash function must be applied to it and, based on the return value *r* of each function (r_1, r_2, …, r_k), the bit with the offset *r* is set to 1. Since there are *k* hash functions, up to *k* bits in the bit array are set to 1 (it might be less because several hash functions might return the same value). The following figure is an example where $m = 16$, $k = 4$ and *e* is the element to be "stored" in the bit array.


$r_1 = h_1(e) = 8 \rightarrow$ set b_8 to 1
$r_2 = h_2(e) = 1 \rightarrow$ set b_1 to 1
$r_3 = h_3(e) = 6 \rightarrow$ set b_6 to 1
$r_4 = h_4(e) = 13 \rightarrow$ set b_{13} to 1
Bit Array
1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15} b_{16}

To check whether an element is "stored" in the bit array, the process is similar. The only difference is that instead of setting *k* bits in the array, it is checked whether any of them are set to 0. If yes, this implies that the element is not "stored" in the bit array.

Let's take a look at an example that illustrates how a bloom filter might be used in practice. An application receives input data at a high throughput. Before processing it, it has to check whether that data belongs to a given set stored in a database table. Note that the rejection rate of this validation is very high. To perform the validation, the application has to call a remote service. Unfortunately, the time needed for the network roundtrip and to actually do the lookup is far too long. Since it is not possible to further optimize it (for example, the network latency is subject to physical limitations), another method must be implemented. In such cases, caches come to mind. The general idea is to duplicate the data stored remotely in a local cache and then to do the validation locally. In this specific case, let's say that it is not practicable. Not enough resources are available locally to store the data. In such a situation, a bloom filter might be useful. In fact, as described above, bloom filters need much less space than the actual set. Since the bloom filter is subject to false positives, the idea is to use it to discard the maximum amount of input data before calling the remote service. As a result, the original validation is not made faster but it is used much less frequently.

A simple implementation in PL/SQL

To provide you with a concrete example, I have written a very simple (in other words, neither space nor time efficient) bloom filter in PL/SQL. The package implementing it provides three routines:

- *init* to initialize the bloom filter
- *add_value* to "store" a string in the bit array
- *contain* to check whether a string is "stored" in the bit array

This is the package specification:

```
CREATE OR REPLACE PACKAGE bloom_filter IS
  PROCEDURE init (p_m IN BINARY_INTEGER, p_n IN BINARY_INTEGER);
  FUNCTION add_value (p_value IN VARCHAR2) RETURN BINARY_INTEGER;
  FUNCTION contain (p_value IN VARCHAR2) RETURN BINARY_INTEGER;
END bloom_filter;
```

Here is the package body (notice that for simplicity I decided to use the package *dbms_random* to implement the hash functions):

```
CREATE OR REPLACE PACKAGE BODY bloom_filter IS
  TYPE t_bitarray IS TABLE OF BOOLEAN;
  g_bitarray t_bitarray;
  g_m BINARY_INTEGER;
  g_k BINARY_INTEGER;

  PROCEDURE init (p_m IN BINARY_INTEGER, p_n IN BINARY_INTEGER) IS
  BEGIN
    g_m := p_m;
    g_bitarray := t_bitarray();
    g_bitarray.extend(p_m);
    FOR i IN g_bitarray.FIRST..g_bitarray.LAST
    LOOP
      g_bitarray(i) := FALSE;
    END LOOP;
    g_k := ceil(p_m / p_n * ln(2));
  END init;
```

```
    FUNCTION add_value (p_value IN VARCHAR2) RETURN BINARY_INTEGER IS
    BEGIN
      dbms_random.seed(p_value);
      FOR i IN 0..g_k
      LOOP
        g_bitarray(dbms_random.value(1, g_m)) := TRUE;
      END LOOP;
      RETURN 1;
    END add_value;

  FUNCTION contain (p_value IN VARCHAR2) RETURN BINARY_INTEGER IS
      l_ret BINARY_INTEGER := 1;
    BEGIN
      dbms_random.seed(p_value);
      FOR i IN 0..g_k
      LOOP
        IF NOT g_bitarray(dbms_random.value(1, g_m))
        THEN
          l_ret := 0;
          EXIT;
        END IF;
      END LOOP;
      RETURN l_ret;
    END contain;
  END bloom_filter;
```

To test the package, a table containing 10,000 rows of 100 bytes each is created.

```
  CREATE TABLE t AS
  SELECT dbms_random.string('u',100) AS value
  FROM dual
  CONNECT BY level <= 10000
```

The bloom filter is initialized with m = 16384. The second parameter (1000) is the number of elements that are expected to be “stored” in the bit array.

```
  SQL> execute bloom_filter.init(16384, 1000)
```

The values of the 1,000 rows are “stored” into the bit array.

```
  SQL> SELECT count(bloom_filter.add_value(value))
    2  FROM t
    3  WHERE rownum <= 1000;

  COUNT(BLOOM_FILTER.ADD_VALUE(VALUE))
  ------------------------------------
                                  1000
```

Now let’s test how many of the 10,000 rows stored in the test table are recognized as belonging to the set according to the bloom filter (remember, false positives are possible).

```
  SQL> SELECT count(*)
    2  FROM t
    3  WHERE bloom_filter.contain(value) = 1;

    COUNT(*)
  ----------
        1005
```

As you can see, there are five false positive. It goes without saying that this number is highly dependent not only on the value of *m* (the number of bits in the array), but also on which hash functions are chosen.

For my simple bloom filter and the same test data as before, the following table shows the number of false positives for different values of *m*.

m	False positives
1,024	6,534
2,048	4,190
4,096	1,399
8,192	220
16,384	5
32,768	0

Bloom filters and Oracle Database

Now that we have seen what bloom filters are, let's describe how Oracle Database makes use of them. As far as I know, they are used in three situations:

- To reduce data communication between slave processes in parallel joins.
- To implement join-filter pruning.
- To support result caches.

The first is available as of Oracle Database 10g Release 2, the other two as of Oracle Database 11g Release 1.

Since I have never investigated the utilization of bloom filters with result caches, and to my knowledge there is no source of information describing it, the next two sections will only describe the other two cases.

Note: No basics about parallel processing and partition pruning are given in this paper. In other words, I expect that you are familiar with them. Information about them is available in the Oracle documentation [4,5] and in my book [6].

Parallel joins

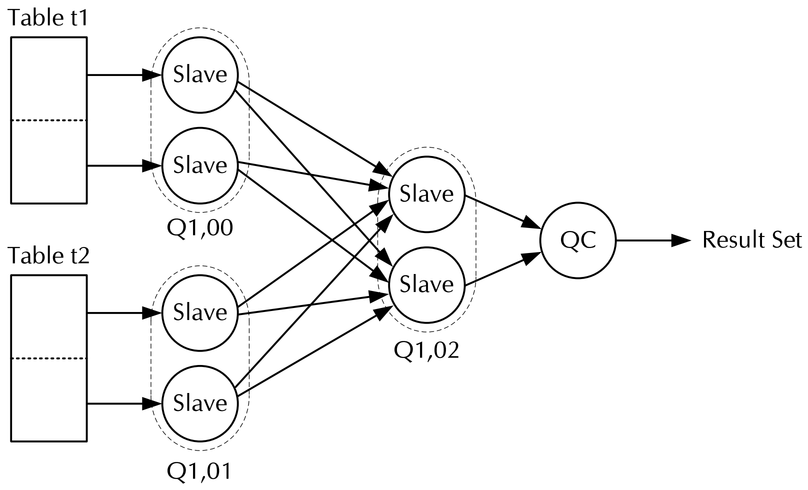
When a hash or merge join is executed in parallel, there are several sets of slave processes that exchange data. For example, in the execution plan associated with the following query, three sets of slave processes are used (each set is identified by a different value in the column TQ).

```
SELECT * FROM t1, t2 WHERE t1.id = t2.id AND t1.mod = 42

-------------------------------------------------------------------------
| Id  | Operation                | Name     |    TQ  |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |        |      |            |
|   1 |  PX COORDINATOR          |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)    | :TQ10002 |  Q1,02 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN BUFFERED    |          |  Q1,02 | PCWP |            |
|   4 |     PX RECEIVE           |          |  Q1,02 | PCWP |            |
|   5 |      PX SEND HASH        | :TQ10000 |  Q1,00 | P->P | HASH       |
|   6 |       PX BLOCK ITERATOR  |          |  Q1,00 | PCWC |            |
|*  7 |        TABLE ACCESS FULL| T1       |  Q1,00 | PCWP |            |
|   8 |     PX RECEIVE           |          |  Q1,02 | PCWP |            |
|   9 |      PX SEND HASH        | :TQ10001 |  Q1,01 | P->P | HASH       |
|  10 |       PX BLOCK ITERATOR  |          |  Q1,01 | PCWC |            |
|  11 |        TABLE ACCESS FULL| T2       |  Q1,01 | PCWP |            |
-------------------------------------------------------------------------

   3 - access("T1"."ID"="T2"."ID")
   7 - filter("T1"."MOD"=42)
```

Simply put, this execution plan is executed in the following way:

- The slave processes of set Q1,00 scan table t1, apply the filter t1.mod = 42, and send the remaining data to the slave processes of set Q1,02 (operations 5 to 7).
- The slave processes of set Q1,02 receive the data and build a hash table (operations 3 and 4).
- The slave processes of set Q1,01 scan table t2 and send the data to the slave processes of set Q1,02 (operations 9 to 11).
- The slave processes of set Q1,02 receive the data (operation 8), probe the hash table and send the rows fulfilling the join to the query coordinator (operations 2 and 3).
- The query coordinator receives the data and returns the result set (operations 0 and 1).

The following figure illustrates the communication between the three sets of slave processes.



The essential thing to notice is that the join is performed by slave processes of the set Q1,02 and, therefore, part of the data sent by the slave processes of the sets Q1,00 and Q1,01 may be discarded because it doesn't fulfill the join condition. In other words, data may be unnecessarily sent to the slave processes of the set Q1,02. The overhead associated with this unnecessary communication is particularly noticeable when two conditions are met. First, when most of the data is discarded because it doesn't fulfill the join condition. Second, when the slave processes, because of RAC, are distributed over several nodes.
To avoid unnecessary communication, Oracle Database may use bloom filters. The following execution plan shows, for the same query as before, such an example.

```
--------------------------------------------------------------------------
| Id  | Operation                | Name     |    TQ  |IN-OUT| PQ Distrib |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |        |      |            |
|   1 |  PX COORDINATOR          |          |        |      |            |
|   2 |   PX SEND QC (RANDOM)    | :TQ10002 |  Q1,02 | P->S | QC (RAND)  |
|*  3 |    HASH JOIN BUFFERED    |          |  Q1,02 | PCWP |            |
|   4 |     PX JOIN FILTER CREATE| :BF0000  |  Q1,02 | PCWP |            |
|   5 |      PX RECEIVE          |          |  Q1,02 | PCWP |            |
|   6 |       PX SEND HASH       | :TQ10000 |  Q1,00 | P->P | HASH       |
|   7 |        PX BLOCK ITERATOR |          |  Q1,00 | PCWC |            |
|*  8 |         TABLE ACCESS FULL| T1       |  Q1,00 | PCWP |            |
|   9 |     PX RECEIVE           |          |  Q1,02 | PCWP |            |
|  10 |      PX SEND HASH        | :TQ10001 |  Q1,01 | P->P | HASH       |
|  11 |       PX JOIN FILTER USE | :BF0000  |  Q1,01 | PCWP |            |
|  12 |        PX BLOCK ITERATOR |          |  Q1,01 | PCWC |            |
|  13 |         TABLE ACCESS FULL| T2       |  Q1,01 | PCWP |            |
--------------------------------------------------------------------------

   3 - access("T1"."ID"="T2"."ID")
   8 - filter("T1"."MOD"=42)
```

Compared to the previous one, there are two additional operations (4 and 11).

- With operation 4, the slave processes of set Q1,02 create the bloom filter named :BF0000 (if there are several bloom filters, the numeric value is increased, i.e. the second one would be named :BF0001).
- With operation 11, the slave processes of set Q1,01 probe the bloom filter :BF0000 before sending data to the slave processes of set Q1,02. Thus, with the exception of false positives, data that doesn't fulfill the join condition is not sent.

The query optimizer decides whether a bloom filter is to be used based on the estimated join selectivity and on the amount of data to be processed. To override these decisions, it is possible to use the hints px_join_filter and no_px_join_filter. However, if the amount of data is small, even by specifying the hint px_join_filter, it is not possible to force the query optimizer to use a bloom filter. To fully disable the feature, the initialization parameter _bloom_filter_enabled must be set to FALSE.

To display information about bloom filters, the dynamic performance view v$sql_join_filter can be used. For example, as shown by the following query, it is possible to check how many rows have been filtered out and probed by the bloom filter. From this we can deduce the number of rows that have gone through the filter (probed – filtered).

```
SQL> SELECT filtered, probed, probed-filtered AS sent
  2  FROM v$sql_join_filter
  3  WHERE qc_session_id = sys_context('userenv','sid');

  FILTERED     PROBED       SENT
---------- ---------- ----------
     81795     100000      18205
```

Be aware that up to Oracle Database 10g Release 2 10.2.0.3, the dynamic performance view v$sql_join_filter returns information only about active bloom filters.

Another dynamic performance view that can show the reduced communication due to a bloom filter is v$pq_tqstat.

Join-filter pruning

The query optimizer is able to perform partition pruning based on literal values, bind variables or join conditions. Up to Oracle Database 10g Release 2, when partition pruning is based on join conditions, the query optimizer can use *subquery pruning* for hash and merge joins. Without going into the details that are explained in my book [6], this type of pruning is of limited usage because part of the query is executed twice. To avoid this double execution, Oracle Database 11g provides another type of partition pruning, *join-filter pruning* (aka *bloom-filter pruning*). The following execution plan shows an example of this new optimization technique (see operations 2 and 5).

```
SELECT * FROM t1, t2  WHERE t1.id = t2.id AND t1.mod = 42

---------------------------------------------------------------
| Id  | Operation                   | Name    | Pstart| Pstop |
---------------------------------------------------------------
|   0 | SELECT STATEMENT            |         |       |       |
|*  1 |  HASH JOIN                  |         |       |       |
|   2 |   PART JOIN FILTER CREATE   | :BF0000 |       |       |
|   3 |    PARTITION HASH ALL       |         |     1 |     8 |
|*  4 |     TABLE ACCESS FULL       | T1      |     1 |     8 |
|   5 |   PARTITION HASH JOIN-FILTER|         |:BF0000|:BF0000|
|   6 |    TABLE ACCESS FULL        | T2      |:BF0000|:BF0000|
---------------------------------------------------------------

   1 - access("T1"."ID"="T2"."ID")
   4 - filter("T1"."MOD"=42)
```

Simply put, this execution plan is executed in the following way:

- All partitions of table t1 are scanned (operations 3 and 4).
- Based on the data returned by operation 3, a bloom filter is created (operation 2).
- Based on the bloom filter, partition pruning on table t2 occurs (operations 5 and 6) and, therefore, only the partitions containing relevant data are scanned.

It is interesting to note that even when there are no restrictions (in the previous case, the restriction is t1.mod=42), the query optimizer might choose this type of partition pruning. This is probably because the overhead of using the bloom filter is so small, that even if no pruning occurs, the overhead associated with its use is negligible.

To disable the feature, the initialization parameter _bloom_pruning_enabled must be set to FALSE.

Conclusion

With bloom filters, Oracle has introduced an optimization technique that can be used in various situations. This paper discussed two of them: parallel joins and join-filter pruning. Both are devoted to improving the performance of very specific types of SQL statements that process large amounts of data. Even if not everyone will be able to take advantage of them, in my opinion, these types of enhancements are very important to keep up with the stringent requirements that ever-larger databases impose. I am sure that in future releases, more and more features like these will be implemented.

References

[1] Bloom, Burton H., "Space/time trade-offs in hash coding with allowable errors"
Communications of the ACM, Volume 13, Issue 7, 1970
http://portal.acm.org/citation.cfm?doid=362686.362692

[2] Knuth, Donald, *The Art of Computer Programming, Volume 3 – Sorting and Searching*
Addison-Wesley, 1998

[3] Zait, Mohamed, "Oracle10g SQL Optimization"
Trivadis CBO Days, 2006

[4] Oracle Corporation, *Oracle Database VLDB and Partitioning Guide, 11g Release 1*
http://download.oracle.com/docs/cd/B28359_01/server.111/b32024.pdf

[5] Oracle Corporation, *Oracle Database Data Warehousing Guide, 11g Release 1*
http://download.oracle.com/docs/cd/B28359_01/server.111/b28313.pdf

[6] Antognini, Christian, *Troubleshooting Oracle Performance*
Apress, 2008
http://top.antognini.ch

About the Author

Since 1995, Christian Antognini has focused on understanding how the Oracle database engine works. He is currently working as a principal consultant and trainer at Trivadis in Zürich, Switzerland. If Christian is not helping one of his customers get the most out of Oracle, he is somewhere lecturing on application performance management. He is a proud member of the OakTable Network and the author of the book *Troubleshooting Oracle Performance* (Apress, 2008).