Bloom Filters for Efficient Coupling between Tables of a Database

Eirini Chioti[1], Elias Dritsas[1], Andreas Kanavos[1], Xenophon Liapakis[3], Spyros Sioutas[2], and Athanasios Tsakalidis[1]

1. Computer Engineering and Informatics Department
University of Patras, Patras, Greece
eldritsas@gmail.com
{chiotie,kanavos,tsak}@ceid.upatras.gr
2. Department of Informatics, Ionian University, Corfu, Greece
sioutas@ionio.gr
3. Interamerican, Greece
liapakisx@interamerican.gr

Abstract. Nowadays, digital data are the most valuable asset of almost every organization. Database management systems are considered as storing systems for efficient retrieval and processing of digital data. However, effective operation, in terms of data access speed and relational database is limited, as its size increases significantly [6]. Bloom filter is a special data structure with finite storage requirements and rapid control of an object membership to a dataset. It is worth mentioning that the Bloom filter structure has been proposed with a view to constructively increase data access in relational databases. Since the characteristics of a Bloom filter are consistent with the requirements of a fast data access structure, we examine the possibility of using it in order to increase the SQL query execution speed in a database. In the context of this research, a database in a RDBMS SQL Server that includes big data tables is implemented and in following the performance enhancement, using Bloom filters, in terms of execution time on different categories of SQL queries, is examined. We experimentally proved the time effectiveness of Bloom filter structure in relational databases when dealing with large scale data.

Keywords: Databases, Bloom Filters, RDBMS, SQL Queries Optimization

1 Introduction

The business data, associated with all of the business activities, are typically stored in relational databases in order to manage them using the SQL language, and more specifically perform SQL queries to the database. The relational databases are particularly effective in their operation. However, their efficiency is limited if they store "big data" with complex correlations [14]. An SQL query can be very expensive in execution cost, and concretely in time and access to

resources, if the execution plan is not optimized. Possible delays in the accomplishment of SQL queries may have impact on application performance using relational databases, thus reducing business performance.

The main way to improve the performance of an SQL query is to reduce the number of required operations/calculations that should be performed during the execution of the corresponding query. However, further reduction of the required commands in an SQL query is not always possible and also requires additional techniques for SQL query performance optimization in a database [5]. In [11], authors investigate this specific problem and recommend the use of IN, EXITS, EQUAL and OPERATOR-TOP along with indexes. Moreover, the bloom filter structure is used in databases such as Google Big Data or Apache HBase in order to decrease searching (in disk) for non-existent records, optimizing in this way the performance of executed SQL queries [3].

The traditional database systems store data in the form of a table with records. Each record corresponds to a different entity object that holds information in a relational table. The relative organization of the databases is effective when there are performing queries on tables with a small number of records. However, as the number of records increases, e.g. hundreds of thousands or millions of records, SQL queries usually search in a much larger number of records in order to locate and access a small number of records or fields [9].

The best way to improve the execution speed of SQL queries in a database is the definition of indexes in fields, which are part of the search criteria of an SQL query. When indexes are not set in a database, then the database management system operates as a reader trying to find a word in a book by reading the entire book. By integrating an index term at the back of a book, the reader can complete the procedure much more quickly. The benefit of using indexes when searching records in a table becomes greater as the number of table entries increases [1]. The role of indexes, in a database is to direct access records according to the search criteria of the SQL query. However, when a table in a database contains millions of records, despite the use of indexes, then the identification of records that meet the search criteria, requires to access thousands of records of the relational table [2]. Therefore, in order to improve the efficiency of execution speed of relational SQL queries, the, in advance, exclusion of a significant number of records that do not meet the search criteria, would be particularly useful. To this purpose, the implementation of Bloom filter structure is suggested; this structure is based on records of the tables and it is further used for the exclusion of records that do not meet the criteria of relevant SQL queries.

The purpose of this research is to examine to what extent the structure of Bloom filter tables in relational databases can affect the performance of data access queries for data tables with millions of records. To achieve the aim of this survey, our contributions lie in the following bullets: (i) implementation of

[1] http://odetocode.com/articles/237.aspx

[2] http://dataidol.com/tonyrogerson/2013/05/09/reducing-sql-server-io-and-access-times-using-Bloom-filters-part-2-basics-of-the-method-in-sql-server

Bloom filter to a relational database, (ii) experimental evaluation of queries with or without the support of Bloom filter and table recording of execution time of queries and (iii) graphic visualization of results to show Bloom filter effectiveness (in terms of integration time) in executing SQL queries on tables with millions of records.

The rest of the paper is organized as follows: in Section 2 the properties and basic components of Bloom filters are introduced. In Section 3, Relational Databases and SQL framework is presented. Moreover, Section 4 presents the evaluation experiments conducted and the results gathered. Ultimately, Section 5 presents conclusions, constraints and draws directions for future work.

2 Bloom Filters Background

2.1 Bloom Filter Elements

The Bloom filter structure, devised by Burton Howard Bloom in 1970, is used for rapid check whether an element is present in a data set or not [1]. It also permits checking if an item certainly does not belong to it. Although the Bloom filters allow false positive responses, the space savings they offer outweigh any downside [8]. A Bloom filter is composed of two parts: a set of k hash functions and a bit vector. The number of hash functions and the length of bit vector are chosen according to the expected number of keys to be added to the Bloom filter and the level of acceptable error rate per case [3].

A number of important components need to be properly defined in order for a bloom filter to operate correctly. These parameters are briefly and comprehensively described in the following paragraphs.

2.1.1 Hash Functions A hash function takes as input data of any length and returns as output an ID smaller in length and fixed in size, which can be employed with the aim to identify elements [4].

The main features that a hash function should have, are the following:

- Return the same value at each iteration with the same data input.
- Quick execution.
- Generate output with uniform distribution in the potential range it produces.

Some of the most popular algorithms for implementing hash functions are: $SHA1$ and $MD5$. These functions differ in safety level and hash value calculation speed. Also, some algorithms homogeneously distribute the values generated by the hash function, but they are impractical. In each case, the selected hash function should satisfy the application requirements.

As for the hash functions number, the bigger this number is, then the hash values are generated in a slower way and the binary vector fills in a faster way.

[3] `https://www.perl.com/pub/2004/04/08/bloom_filters.html`
[4] `https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff`

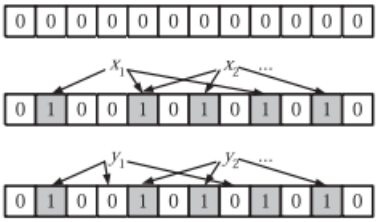


Fig. 1: Bloom Filter Overview

However, this decision increases the incorrect predictions on the existence of an object in a dataset [5]. The optimal number of hash functions derives from the following formula in [8]:

$$k = \frac{m}{n} \ln(2) \tag{1}$$

where m is the binary vector length and n the number of inserted keys in bloom filter. When selecting the number of hash functions to be used, we also calculate the probability of false positive predictions. The previous step is repeated until we get an accepted value for the probability index of false positive responses [4].

2.1.2 Binary Vectors Length The length of the binary values of a Bloom filter vector affects the pointer value of false positive responses of the filter. The greater the length of the binary vector values, the lower the probability of false positive responses. Conversely, as the length of the vector is shrinked, the relative probability is increased. Generally, a Bloom filter is considered complete when 50% values of bits in the array are equal to 1. At this point, further addition of objects will result in the increase of false positive responses rate [10].

2.1.3 Key Insertion We initialize a Bloom filter by setting the values of binary vector equal to 0. To insert a key into a Bloom filter, the relevant k hash functions are originally performed and positions of the binary vector, which corresponds to hash values, change from 0 to 1. If the relevant bit is already set to 1, then the value of the relevant bit does not further alter [6]. Each bit of the vector can simultaneously encode multiple keys, which makes the Bloom filter compact as shown in Figure 1 [2].

The overlapping values do not permit a key removal from the filter, since it is not known whether the relevant bits are not activated by other key values. The only way to remove a key from a Bloom filter is to rebuild the filter from scratch, thus not incorporating the key to be removed from the Bloom filter. For

[5] `https://llimllib.github.io/bloomfilter-tutorial`
[6] `https://www.perl.com/pub/2004/04/08/bloom_filters.html`

checking the possibility for a key in the Bloom filter to be present, the following procedure is applied. Initially, the hash functions are applied to the search key, and then we check the relevant bits generated by the hash functions to be all activated. Concretely, if at least one of the bits is disabled, it is certain that the corresponding key is not included in the filter. If all bits are turned on, then we know that with high probability, the key has been introduced.

2.2 Space-Time Advantages and Constraints

The implementation of a Bloom filter is relatively simple in comparison with other relevant search structures. In addition, the use of a Bloom filter ensures the fast membership checking of a value and in following absolute reliability of the non-existence of an object in it (no false negatives) [13]. Concerning the time required for adding a new item or to control whether a point belongs to a set of data, it is independent of the number of elements in the filter [7]. More to the point, a strong advantage of Bloom filters is the storage space saving in comparison with other data structures such as sets, hash tables, or binary search trees.

The insertion of an element into a Bloom filter is an irreversible process [8]. The size of data in a Bloom filter must be known in advance for determining the vector length and the number of hash functions. However, the number of objects that will be imported into a Bloom filter are not always known in advance. It is theoretically possible to define an arbitrarily large size, but it would be wasteful in terms of space and would overturn the main advantage on the Bloom filter, which is storage economy. Alternatively, a Dynamic Bloom filter structure could be adopted, which, however, is not always possible. There is a variant of the Bloom filter, called Scalable Bloom filter, which dynamically adjusts its size for different number of objects. The use of a relative Bloom filter could alleviate some of its shortcomings. A Bloom filter cannot produce the list of items imported, but it can only check whether an item has been introduced in a dataset. Finally, the Bloom filter cannot be used for answering questions about the properties of the objects.

3 Bloom Filters and RDBMS

3.1 Relational Database Management Systems

The relational database management systems have been a common choice for storing information in databases used for a wide range of data such as financial, logistic information, personal data, and other forms of information, since 1980. The relational databases have replaced other forms such as hierarchical or

[7] `https://prakhar.me/articles/bloom-filters-for-dummies`
[8] `http://bugra.github.io/work/notes/2016-06-05/a-gentle-introduction-to-bloom-filter`

network databases, as they are easier in understanding and their use is convenient. The main advantage of relational data model is that it allows the user to make query-in data access command, without the need to define access paths to stored data or other additional details [7]. Furthermore, the relational databases keep their data in form of tables. Each table consists of records, called tuples, and each record is uniquely identified by a field, i. e. primary key, which has a unique value. Each panel is usually connected to at least another database table in relation to the form: (i) one-by-one, (ii) one-to-many, or (iii) many-to-many.

These relationships grant users unlimited ways of data access and dynamic combination amongst them from different tables. Nowadays, the market provides more than one hundred RDBMS systems and the most popular of them are the following: (i) Oracle, (ii) MySQL, (iii) Microsoft SQL Server, (iv) PostgreSQL, (v) DB2 and (vi) Microsoft Access (DB-Engines 2016), etc. [9].

The SQL language is used for user communication with a relational database [12]. An SQL query demands no knowledge of the internal operation of database or the relevant data storage system [15]. According to ANSI (American National Institute Standards) standards, the SQL is a standard language for relational database management systems. Moreover, the SQL language is used in order to query a database for the management of such data and also for the data update or retrieval from a database. Some examples of relational databases that use SQL are: Oracle, Sybase, Microsoft SQL Server, Access and Ingres.

The most important commands of SQL query language are [10]: SELECT, UPDATE, DELETE, INSERT INTO, CREATE DATABASE, ALTER DATABASE, CREATE TABLE, ALTER TABLE, DROP TABLE, CREATE INDEX, DROP INDEX.

The SQL commands are classified into the following basic types:

- **Query Language with key command:** where the Select command for accessing information from the database tables is used.
- **Data Manipulation Language with key commands:** (i) Insert-introduction of new records, (ii) Update-modify records, and (iii) Delete-delete records.
- **Data Objects Definition with key commands:** (i) Create Table, and (ii) Alter Table.
- **Safety Control of Database with key commands:** (i) Grand, Revoke for user rights management to database objects, and (ii) Commit, Rollback for transactions management.

3.2 Queries Language-SQL

3.2.1 Membership Queries The command SQL IN controls whether an expression matches any value from a list of values. Furthermore, it is used in order to prevent multiple use of the OR command in SELECT, INSERT, UPDATE or DELETE queries [11]. Besides checking should an expression belong to a set

[9] `http://db-engines.com/en/ranking/relational+dbms`
[10] `http://www.w3schools.com/sql/sql_syntax.asp`
[11] `https://www.techonthenet.com/sql/in.php`

of values registered directly to a relevant query SQL, it may also check if an expression is part of a set of values from other tables.

3.2.2 Join Queries The union queries, which combine values from two or more data tables based on a JOIN criterion, usually concern relationships between relevant tables. More to the point, JOIN queries are distinguished in four categories:

1. **Inner Join:** returns the values from Table A and Table B that satisfy the joining criteria.
2. **Left Join:** returns all the values from Table A and the values of the Table B meeting the joining criteria.
3. **Right Join:** returns all the values from Table B and the values of the Table A that meet the joining criteria.
4. **Outer Join:** returns all the values from Table A and Table B regardless if they satisfy the relevant criteria combination.

3.2.3 Exist Queries The existence control queries are used in conjunction with a secondary query. It is considered that the control condition is satisfied when the secondary query returns at least one relevant registration. The verification can be used in terms of the following queries: SELECT, INSERT, UPDATE or DELETE [12].

3.2.4 Top Queries The command TOP limits the number of records that a query will return, that is to a specified number of rows or a specified percentage of records from the 2016 version of SQL Server [13]. When the command TOP is used in combination with the ORDER BY command, then the first N records are returned according to the sorting arrangement provided by the ORDER BY command. Otherwise, N unsorted records are returned.

In addition, the TOP command specifies the number of records returned by a SELECT statement or affected by a plethora of command statements, such as INSERT, UPDATE, JOIN, or DELETE. The TOP SELECT command can be particularly useful in large tables with thousands of records. The access and choice of a large number of records can adversely affect the performance execution of a query.

3.3 Indexes Table

Indexes are auxiliary structures in a relational database management system with the aim of increasing data access performance to the database. Relevant helping structures are created in one or more fields (columns) of a table or a

[12] `https://www.techonthenet.com/sql/exists.php`
[13] `https://docs.microsoft.com/en-us/sql/t-sql/queries/top-transact-sql`

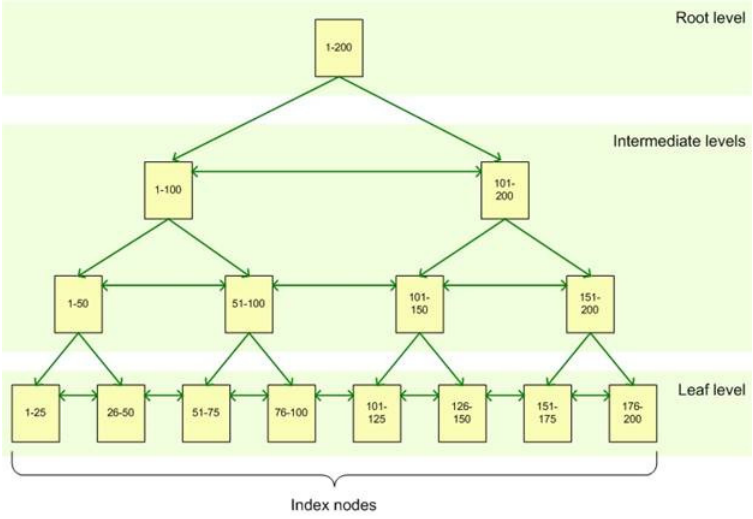


Fig. 2: B-Tree overview

database. Moreover, an index provides a quick way to search data based on the values in the specific fields that are part of the index.

For example, if an index on the primary key of a table is created, and then a series of data based on the values of the corresponding fields is found, then the SQL Server finds the value of the index field first and in following it uses the relevant index so as to quickly locate the whole relevant table entries. In this way, without the index marker field, it would require a scan of the entire table line by line, directly influencing the performance of the relevant query execution [14].

Furthermore, an index consists of a set of pages that are organized into B-tree data structure. The relevant structure is hierarchical, comprising a root node at the top of the tree and the leaf nodes at the lower level, as illustrated in the above Figure 2. When a query, including a search criterion, is executed, then the query starts delving into relevant records from the root node and navigates through intermediate nodes, which are the leaf nodes of the B-tree structure. After locating the relevant leaf node, the query will access the interrelative record either directly (in the case of clustered index), or through a pointer to the relevant data record (if it is a non clustered index).

A table in an SQL Server database can have at most one clustered index and more than one non clustered index, depending on the version of SQL Server that is used.

[14] `https://www.simple-talk.com/sql/learn-sql-server/sql-server-index-basics`

4 Experimental Evaluation in SQL Server

In this section, the results of the experiments conducted in the context of this research in order to evaluate the use of Bloom filters, are presented. We perform a series of common SQL database queries with and without the support of the Bloom filter and graphically present the resulted time performance of executed SQL queries. The SQL queries utilized are the following: *In*, *Inner Join*, *Left Join*, *Right Join*, *Exists* and *Top*.

The following Tables 1 and 2 as well as Figure 3 show the execution times of the questions described previously. In particular in the corresponding Tables, the results are shown with and without the use of Bloom filter by introducing the label BF as the relative number of records is changed.

Table 1: SQL Queries Execution Time Results vs Data Size

Execution Time in seconds						
Data	**In**	**In BF**	**Inner Join**	**Inner Join BF**	**Left Join**	**Left Join BF**
10.000.000	44	24	44	24	1	1
9.000.000	38	24	41	26	1	1
8.000.000	26	21	26	24	1	1
7.000.000	19	21	19	20	0	1
6.000.000	19	20	19	20	0	1
5.000.000	18	19	19	19	0	1
4.000.000	13	14	13	12	0	1
3.000.000	11	12	12	13	0	1
2.000.000	10	12	11	12	0	1
1.000.000	3	3	3	3	0	1

For all SQL commands and in particular for small number of records, we observed that the adoption of Bloom filter structure overloaded the system and thus, the execution of the queries without the use of Bloom filter is much faster.

As the number of table records is increased and especially for values more than (or equal to) $8,000,000$, the performance advantage offered by employing the Bloom filter structure increases significantly and the difference in speed execution of queries is obvious as it also rises exponentially.

It is important to consider the fact that during the repetitive execution of the same queries, we observed the same runtime, but sometimes there was a gap of about two seconds between results. In these cases, we decided to take the average values in the relevant cases.

Table 2: SQL Queries Execution Time Results vs Data Size

Execution Time in seconds						
Data Size	**Right Join**	**Right Join BF**	**Exists**	**Exists BF**	**Top**	**Top BF**
10.000.000	44	42	43	23	26	6
9.000.000	37	27	38	24	18	6
8.000.000	26	26	25	25	7	6
7.000.000	19	20	18	20	7	5
6.000.000	19	20	19	20	7	5
5.000.000	19	19	19	19	5	5
4.000.000	13	12	12	12	5	5
3.000.000	12	13	11	12	5	5
2.000.000	11	12	12	12	3	3
1.000.000	3	3	2	3	1	2

5 Conclusions

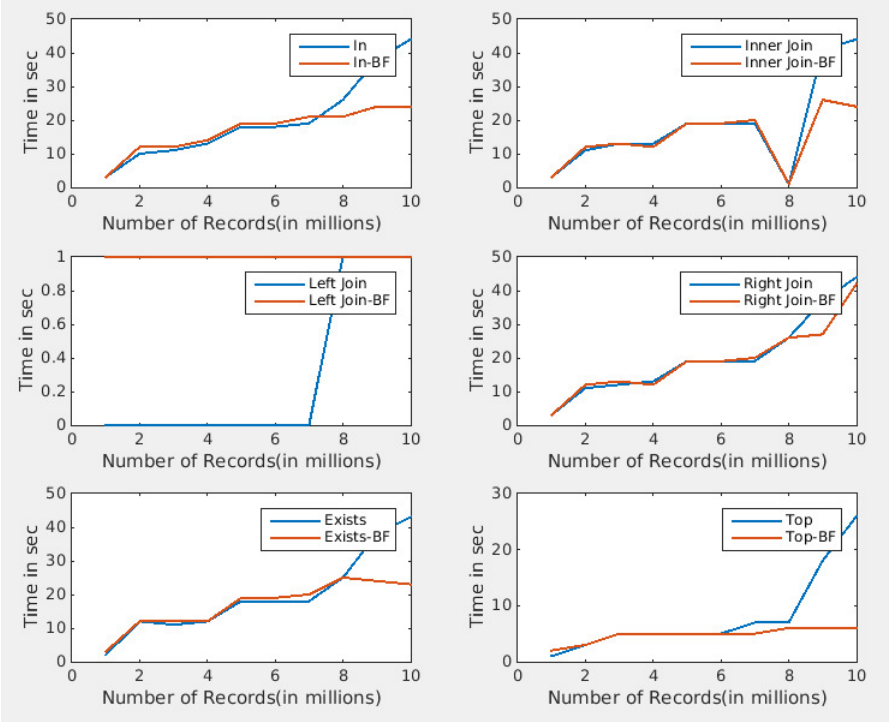
5.1 Research Conclusions

The large response times of SQL queries in relational databases affects not only the users, but also other applications that may run on the same computer or the network itself hosting the relevant database. The Bloom filter, capacity wise, is an effective solution and it has been used in numerous applications in the past, especially when immediate control of an object membership was required.

The relevant experiments suggest that the inclusion of Bloom filter structure in an SQL Server database (with large number of records - $10,000,000$ records) that may increase its data access performance. The optimization of query execution time to a database, using Bloom structure, allows users to quickly extract the needed information and increase the efficiency of relevant database. The Bloom structure in a relational database acts as a filter that removes from the join, membership or existence control queries the need to access-process records that do not meet the criteria of relevant questions. The potential profit from this restriction involved in accessing-searching records through an SQL query highly depends on the false positive records that the control of the Bloom filter returns. This relative number is limited as the length of the binary values of Bloom filter is increased.

In this topic, an acceptable speed execution as well as balanced storage requirements, according to the requirements of each database instance and the user requirements, which concern the access speed of the relevant database, should be chosen. Especially, in cases like a database containing historical data records with no probability of further record updates, the adoption of Bloom filter for faster access among numerous relevant tables can be considered a solution that could lead to increased efficiency.

Fig. 3: Queries Execution Time vs Records Size



As it can be seen from the execution times of SQL queries (Tables 1, 2 and Figure 3), the benefit of the, in advance, restriction of records involved in an SQL query is greater than including all records of data tables and indexes used for direct access to them. It should be noted that as the experimental measurements show, the application of Bloom filter structure in a database deserves to be selected only when the number of entries in the relevant tables are very large. Consequently, the use of Bloom filter may have the opposite effect, i.e. increase the query runtime.

5.2 Research Constraints

In the evaluation of the Bloom filter, we did not take into account possible delays caused by maintenance and regular updates of the Bloom filter structure during the record updates in the relevant tables. These possible delays could be caused in the execution of other SQL queries as well. Although all experiments were performed on the same machine, the ones with Bloom filter that were performed at different times, may have been affected (in performance) by possible processes running in the background. These relevant deviations do not directly affect the performance comparison among the same queries with or without the use of Bloom filter, but mostly between different SQL commands used.

5.3 Future Extensions

A promising and useful step would be to investigate the applicability of Bloom filters in other relational database management systems (like Oracle, SysBase, MySQL) with the aim of generalizing previous conclusions drawn from experimentation on the SQL Server relational database management system. Also, a possible review of the actual performance of database operations with millions of records used to store application data, will allow more reliable conclusions about the use of Bloom filter structure in relational databases. Thus possible delays of the system during the application operation can be taken into account.

References

1. Blustein, J., El-Maazawi, A.: Bloom filters:A Tutorial, Analysis and Survey (2002)
2. Broder, A.Z., Mitzenmacher, M.: Survey: Network applications of bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2003)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26(2), 4:1–4:26 (2008)
4. Christensen, K.J., Roginsky, A., Jimeno, M.: A new analysis of the false-positive rate of a bloom filter. Information Processing Letters 110(21), 944–949 (2010)
5. Gupta, M.K., Chandra, P.: An empirical evaluation of like operator in oracle. BVICAM's International Journal of Information Technology 3(2) (2011)
6. Khan, M., Khan, M.N.A.: Exploring query optimization techniques in relational databases. International Journal of Database Theory and Application 6(3), 11–20 (2013)
7. Kim, W.: On optimizing an sql-like nested query. ACM Transactions on Database Systems (TODS) 7(3), 443–469 (1982)
8. Kirsch, A., Mitzenmacher, M.: Less hashing, same performance: Building a better bloom filter. In: Annual European Symposium on Algorithms (ESA). pp. 456–467 (2006)
9. Larson, P., Clinciu, C., Hanson, E.N., Oks, A., Price, S.L., Rangarajan, S., Surna, A., Zhou, Q.: SQL server column store indexes. In: ACM SIGMOD International Conference on Management of Data. pp. 1177–1184 (2011)
10. Lyons, M.J., Brooks, D.M.: The design of a bloom filter hardware accelerator for ultra low power systems. In: International Symposium on Low Power Electronics and Design. pp. 371–376 (2009)
11. Oktavia, T., Sujarwo, S.: Evaluation of sub query performance in sql server. EPJ Web of Conferences 68 (2014)
12. Ramakrishnan, R., Donjerkovic, D., Ranganathan, A., Beyer, K.S., Krishnaprasad, M.: SRQL: sorted relational query language. In: International Conference on Scientific and Statistical Database Management (SSDBM). pp. 84–95 (1998)
13. Roozenburg, J.: A literature survey on bloom filters. Research Assignment in Computer Science (2005)
14. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: A data provenance perspective. In: ACM Southeast Regional Conference. p. 42 (2010)
15. Winand, M.: SQL Performance Explained: Everything Developers Need to Know about SQL Performance (2012)