A Novel Paging Query Optimization Technique for Relational Databases

Muhammad Naeem Ahmed Khan
Shaheed Zulfikar Ali Bhutto Institute of Science and Technology (SZABIST), Islamabad, Pakistan

Database systems are designed to store data in structured formfor efficient retrieval and processing. SQL fetches data from these databases in the form of pages each consisting of several rows. Paging query has direct impact on the systemperformance. The major bottleneck for large databases that affect paging query is the volume of data. Since a full table scan is involved in this process so a significant transfer time is required to move data between storage media and database server. We propose an optimized solution for paging query based on key technologies of paging query efficiency. The proposed solution for SQL optimization is based on the concept of offloading. Contrary to the typicaldatabase systems, only the data sought by the client is returned to the database instance from storage. ShiftingSQL processing off the database server eliminates massive amount of futileI/O transfers. Hence, queries run much faster and are processed in a shorter timespan. The experimental results show that the proposed solution is effective and efficient.

Keywords: Query Optimization, Paging Query, Relational Database Management Systems, Real Application Cluster, Process Global Area.

1. INTRODUCTION

Data in an organization is generally considered a more valuable asset than the hardware and software resources. Databases pagination is desired by web developers particularly when they deal with larger volume of data (Sun and Wang, 2011). Query optimization results in minimal response time and maximum throughput (Khan and Khan, 2013; Ali Khan and Khan, 2018). Query processing involves three stages: decomposition, optimization and execution. Decomposition parses and binds the query by checking its syntax, semantics and authorization and builds an algebraic tree. In the optimization stage, algebraic tree is used by the query optimizer to produce the best optimization plan. The optimization plan is then executed in the execution phase and the algebraic tree becomes a physical operator tree. In a relational database management system (RDBMS), query optimizer receives the parsed tree executed by the execution engine using the mapped physical operators and produces an optimized cost effective plan in terms of less resource consumption. Query optimizer selects the most appropriate execution plan which is fed to query execution engine for execution (Chaudhuri, 1998). Various methods are used in RDBMS to process and optimize queries according to their complexity. Whence the amount of data increases, complex queries are used to fetch data which may consist of large number of joins, sub-queries, groups etc. (Bellamkonda, Ahmed, Witkowski, Amor, Zait, and Lin, 2009). Generally, SQL returns results page by page, e.g. 40-60 records at a time. An ordered set of rows evaluated by a query is called pagination. Paging query process normally does not encounter a problem in case of small data size, but if data volume is huge then more client resources are required to cache data, which eventually degrades performance of the application and reduces its availability. To overcome this problem, a better optimized solution is needed for paging query process. Normally, resources at database server are enough to process the query. If the required memory management features such as System Global Area (SGA) and Process Global Area (PGA) are used in query optimization, then it would lead to elevated application performance (Sun and Wang, 2011). Queries involving disk access are often inefficient as it is necessary to cache data into memory prior to processing it. Loading large volume of data into memory can take longer, and for larger data volumes, it is not practicable to cache everything into the memory of DB server (Bach, Arao,

Colvin, Hoogland, Osborne, Johnson, and Poder, 2015). The storage should be smart enough to recognize the queries involved in full table scan. Taking SQL processing off the database server frees server CPU cycles and eliminates the massive amount of unproductive I/O transfers. These freed resources can be used to entertain other user queries effectively (Bach et al., 2015). (Lohman, 2014) argue that query optimization is not yet a solved problem. A novel method for query optimization in the cloud computing particularly for multi-tenant databases based on joining indexes is proposed in (Eidson and Collins, 2016).

2. BACKGROUND AND RELATED WORK

Query optimization involves rewriting where a query is transformed into an efficient query. The next stage is planning where planner module performs different search mechanisms for efficient plans exploration (Ioannidis, 1996). The Logical operator trees in the algebraic space are mapped to the physical operators or available indexes e.g. merge scan and nested loops. (Sun, Zhao, and Ge, 2009) describe query optimization for speedy data retrieval by using indices. Database design should be correct and database tables be in 3rd normal form (3NF) for better database performance. Tuning of the logical database structure can be done by the indices techniques if database design is made 3NF.Proper indexing limits I/O operation and improve query response time. Query performance is affected when correlated nested queries exist. (Li, Han, and Ding, 2010) discuss scenarios of sub-queries when objects and conditions of outer and inner query are the same a mechanism called intra-query redundancy. However, sub-query is occasionally needed when aggregate calculation is involved. (Gupta and Chandra, 2011) focus on the effects of LIKE operator. Oracle uses two types of optimizers to handle queries: cost-based and rule-based optimizer. The cost-based optimizer focuses on cost of the query to deliver maximum throughput and minimum response time. The rule-based optimizer generates evaluation plans for the query. Cost-based and heuristic based approaches are usually used for query transformations. (Bellamkonda et al., 2009) describe various query transformation techniques like group by, sub-query, un-nesting, semi joins and anti joins. (Herodotou, Borisov, and Babu, 2011) discuss SQL query optimization in terms of partition tables. The proposed technique can be used to choose the efficient execution plan as partitioned tables offer variety of advantages like query pruning, parallel data access etc. The distinctiveness of grid systems is the flexibility and power which make it an efficient platform for distributed processing. (A. and F., 2009) explains query optimization for centralized DB and grid environment. (Zafarani, Feizi Derakhshi, Asil, and Asil, 2010) introduce an optimization method for distributed databases by reducing the join orders. Weights of queries in terms of frequent accesses are calculated and the queries having high score correspond to frequent access, and are kept in the database memory for efficient pagination. (O'Neil and Graefe, 1995) discuss query optimization in OLAP and decision support systems. The study executes queries with multiple joins by focusing on star join technique which entails bitmap indexes.This approach avoids full table scan and produces good evaluation plan to improve the performance. (Bruno, Chaudhuri, and Ramamurthy, 2009) introduce query hints to change the default behavior of the optimizer for obtaining better execution plan. Query hints enforce optimizer to follow rules written in the query hints to pick a better query plan.

Motivation: Traditionally, paging query process at database server fetches data and delivers the resultant rows to the client cache. For small data size, this process work fine but when volume of data increases then application performance may degrade as more client resources are needed to cache data. Another bottleneck that affects paging query is volume of the data returned to database server from the storage media.To address this issue, an optimized solution is required for the paging query process as it is not sensible to cache all the data in database servers memory.

3. PROPOSED SMART SCAN MODEL

Our proposed technique for paging query optimization mainly addresses the issue of improving paging query efficiency for large volume of data. Another issue we address is to increase paging

query efficiency by slashing data transfer time between storage media and the database servers. Our technique describes the essential steps to perform paging query in an optimized way. We choose Oracle 11 ver11.2.0.4 as backend database as it runs on all flavors of Unix/Linux and Windows. However, the proposed steps of our technique are generic and can be applied to any RDBMS. We propose smart scan/offloading concept for DB optimization. The key steps of the proposed technique are described below.

3.1 Database Optimization

For DB optimization, we tune instance memory (SGA, PGA), enable AMM, adjust Disk I/O, utilize indexes and reduce swapping. The following configurations are recommended to be optimized.

Tuning SGA: The SGA is a shared memory segments allocated by the OS when an instance starts and it is revoked when instance terminates. This segment is shared by all the foreground and background processes. Certain SGA parameters can be resized/tuned by making changes in spfile. The following three areas of SGA should be tuned to optimize the query.

DB Buffer Cache Optimization: Buffer cache is a place where query results fetched from the DB are stored. As long as the data blocks remain in buffer cache, the query results are sent to the user quickly when user requests for the same data. Size of DB buffer cache should be sufficiently large to hold frequently accessed data to minimize disk I/O. Oracle uses Least Recently Used (LRU) algorithm to page out data blocks from the memory. Oracle allows resizing DB buffer cache dynamically without restarting the DB. The DB buffer cache should be resized to get cache hit ratio above 90% and caches free ratio close to zero.

Log buffer Optimization-Default: Log buffer is a small staging area in memory that holds all the requested changes to the DB. All these changes are written to the redo log files which are used for recovery purposes. Log buffer size should not be large since redo generation limits the DB performance as no DML (Data Manipulation Language) command can be used until redo log writing finishes.

Shared Pool Optimization: Shared pool is a complex structure of SGA and we discuss four of its important structures. The library cache stores the recently executed SQL query in the compile form. When SQL statement is used for the very first time then it is parsed before execution. Afterwards, it is executed without reparsing if the same statement is reused. The data dictionary cache stores object definitions and other metadata to make object parsing faster. To avoid repeated reading from data dictionary, these objects are cached into PL/SQL area. Shared pool size is indispensable for performance so it needs to be large enough to hold frequently executed code and object definitions. An undersized shared pool can degrade performance due to repeated parsing of the statements each time. On the contrary, the oversized shared pool also negatively impacts DB performance as it takes more time to search it. Shared pool can be resized dynamically by altering shared_pool_size parameter. Shared pool uses LRU algorithm to replace the objects not used for a long time. Normally, hit ratio values of data dictionary cache and free ratio value should be more than 90% and 20% respectively.

Tuning PGA: PGA is allocated during the process creation and is revoked when the process terminates. PGA contains data as well as the control information for a foreground or background process. It is recommended to use PGA for sorting purposes. The PGA has a sort area, which is used to perform sorting efficiently as compared to temporary tablespaces which needs disk I/O operations. Usually sort area hit ration should be over 95%.

Enabling Automatic Memory Management: Automatic shared memory management in Oracle 11g is achieved using the parameter MEMORY_TARGET. When AMM is enabled then cache sizes of all the pools are readjusted according to existing workload.

Adjustment of Disk I/O: Disk IO is a vital factor that affects the query performance. It is recommended to distribute DB files at different disks to achieve disk load balancing and avoid disk competition. For smooth load across all the disks, distribute these data files evenly across different disks if these belong to the same tablespace. Always create separate tablespaces for

tables and indexes. Distribute the data files and index files of these table-spaces at different disks to avoid disk competition. It is suggested to store SYSTEM tablespace and application tablespace on different disks to avoid disk competition.

Utilizing Indexes: Indexes are used to avoid the full table scan. Excessive number of indexes can degrade performance. The frequently used columns or the columns that appear in joins are best candidates for indexing.

3.2 SQL Optimization

Along with database optimization, it is necessary to tune SQL/PLSQL code as well. Usually, we want SQL query to return data page by page e.g. 60 records at time. Sorting plays an important role in this situation. Data filtering at appropriate layer for a complex query can lead towards better query performance. In Figure 1, data filtering is done at outermost layer.

```
Select * from ( select ROWNUM row_num, SQ2.* from
                        (select * from test where col is not null order by col) SQ2
        )SQ1
where row_num between given_start_rownum and given_end_rownum;
```

Figure 1. Filtering at outermost Layer.

Better efficiency can be realized if filtering is applied at intermediate layer (Figure 2).

```
Select * from ( select ROWNUM row_num, SQ2.* from
                        (select * from emp where sal is not null order by sal) SQ2
where row_num < given_end_rownum
        )SQ1 where row_num > given_start_rownum;
```

Figure 2. Filtering at intermediate Layer.

For simplicity, we use category pagination where the rows are paged from a single table. We used SQL developer to trace out the efficiency and the query plan of a query using different optimization methods.

3.3 Smart Scan/Offloading

Smart scan model address major bottlenecks on the large databases that affect paging query i.e. transfer time of huge amount of data between storage systems and DB servers. It is observed that queries involving disk access are often inefficient. Besides the classic intelligence that resides on the DB server, the storage should also be intelligent enough to recognize queries involved in full table scan. After the scan, instead of returning huge volume of data, only the filtered data should be returned. Shifting SQL processing off the DB server frees server CPU cycles and eliminates the enormous amount of unproductive I/O transfers. In case of large and complex queries, the predicate filters all rows in the table. Whether the filtered rows are less or more, all the table blocks are read. These blocks are then sent to the storage network and are loaded into cache (memory). Hence, the records read into memory are more than what the client has required in the predicate filter to complete the SQL operation. This causes a huge amount of unproductive I/Os. The database operations are handled differently in the smart scan model. The storage is intelligent enough to recognize queries involving in the table scans. The storage uses a direct read mechanism for smart scan processing. Queries that perform table scans are processed in the storage layer which returns only the filtered data to the database instead of the

whole table data. The storage server uses special functions like row-filtering, join-processing and column-filtering. Suppose a user issues a SELECT statement with row-filtering. The database kernel finds the required data and constructs an iDB command that reflects the SQL command being issued by the client. The kernel sends that command to the storage which is processes it. The storage layer returns the iDB message containing the required rows and columns to the database instance. This result is not stored in the buffer cache of DB server as these are not the block images and are already filtered. These rows are then returned to the client. The smart scan, also termed as offloading, is used to solve the problem of extra time spent on moving irrelevant data from storage tier to database tier. This concept meets three goals: reduce data volume being transferred between the DB and storage tiers, reduce CPU cycles and reduce disks accesses (i.e., I/Os). The classical and smart scan processing is shown in Figure 3. The data flow mechanism of both the models is shown alongside to highlight the disparities in terms of efficiency and responsiveness. The proposed scheme is adaptive and kicks-off even if size of the data is enormous and the queries produce optimal results for a typical database.

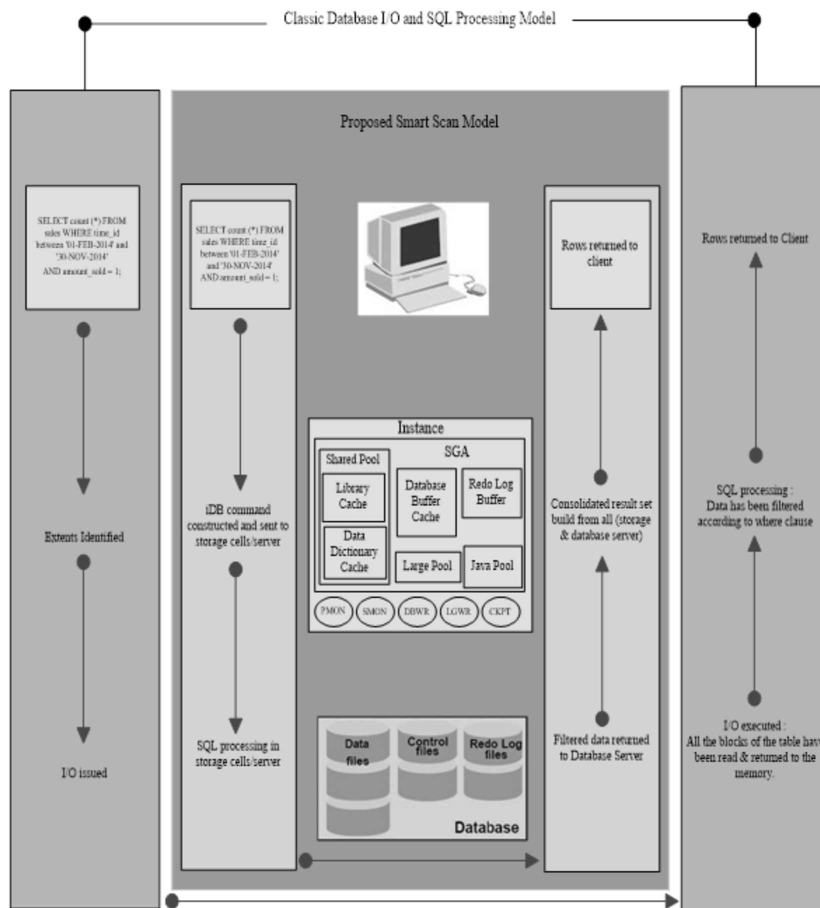

Figure 3: Classic Database Model and the Proposed Smart Scan Technique.

4. BACKGROUND AND RELATED WORK

We performed experiments to test our proposed technique. We used Oracle 11.0.2.4 as backend database on Exadata platform. To evaluate the performance of our technique, we developed

multiple queries and executed these. Performance was measured through statistical results. The detailed description of these queries is described below:

Query 1: To disable the Smart Scan, we used hint for the optimizer in the query. Hint enforces the optimizer not to use the Smart Scan and is enabled by default. The query shown in Figure 4 does the full table scan. The dates are selected randomly and for this instance it returns 12044 records.

Query 2: After executing query with hint, the statistics were examined and it became evident that all the data processed by the query (physical read total bytes) is returned to the database server over the storage network (cell physical IO interconnects bytes). Results are shown in Figure 5.

Query 3: After resetting the statistics again, the same count(*) query was executed but this time no hint was used so that it uses the Smart Scan capabilities. Query results are shown in Figure 6.

Query 4: We executed the Statistics query again and noted that only 228KB data was returned to the DB server (cell physical IO interconnect bytes) out of 559 MB of I/O (physical read total bytes). This was due to Smart Scan. Figure 7 shows that Smart Scan acts on all the I/O related with the query. All the I/O associated with the query were performed at storage layer/server and returned only 228KB. The experimental results obtained by executing these queries for different datasets are shown in Table 1, 2 and 3.

```
MAJID @ PCRMPROD1 > select /*+ OPT_PARAM('cell_offload_processing' 'false') */
                      2  count(*) from sales
                      3  where time_id between '01-FEB-2013' and '30-NOV-2013'
                      4  and amount_sold = 1;

               COUNT(*)
-------------------------
                  12044

MAJID @ PCRMPROD1 >
```

Figure 4: Using Query hint to disable Smart Scan

```
MAJID @ PCRMPROD1 > select a.name, b.value/1024/1024 MB
                      2  from v$sysstat a, v$mystat b
                      3  where a.statistic# = b.statistic#
                      4  and (a.name in ('physical read total bytes',
                      5               'physical write total bytes',
                      6               'cell IO uncompressed bytes')
                      7  or a.name like 'cell phy%');

NAME                                                                      MB
------------------------------------------------------------   -------------
physical read total bytes                                         695.155877
physical write total bytes                                                 0
cell physical IO interconnect bytes                               695.155877
cell physical IO bytes saved during optimized file creation                0
cell physical IO bytes saved during optimized RMAN file restore            0
cell physical IO bytes eligible for predicate offload                      0
cell physical IO bytes saved by storage index                              0
cell physical IO bytes sent directly to DB node to balance CPU             0
cell physical IO interconnect bytes returned by smart scan                 0
cell IO uncompressed bytes                                                 0

10 rows selected.

MAJID @ PCRMPROD1 >
```

Figure 5: Results without Smart Scan

```
MAJID @ PCRMPROD1 > select a.name, b.value/1024/1024 MB
                  2  from v$sysstat a, v$mystat b
                  3  where a.statistic# = b.statistic#
                  4  and (a.name in ('physical read total bytes',
                  5              'physical write total bytes',
                  6              'cell IO uncompressed bytes')
                  7  or a.name like 'cell phy%');

NAME                                                                  MB
--------------------------------------------------------------------  -------------
physical read total bytes                                             695.155877
physical write total bytes                                            0
cell physical IO interconnect bytes                                   .232566257
cell physical IO bytes saved during optimized file creation           0
cell physical IO bytes saved during optimized RMAN file restore       0
cell physical IO bytes eligible for predicate offload                 695.155877
cell physical IO bytes saved by storage index                         0
cell physical IO bytes sent directly to DB node to balance CPU        0
cell physical IO interconnect bytes returned by smart scan            .232566257
cell IO uncompressed bytes                                            695.155877

10 rows selected.

MAJID @ PCRMPROD1 >
```

Figure 6: No query hint to use Smart Scan

Table I: Comparison table between Classic SQL processing and Smart Scan processing

Physical read total bytes (MB)	**Classical SQL processing**	**Smart Scan processing**	**Unproductive I/O bytes to be eliminated**
559.039063	559.039063	0.2222244263	558.816818737
559.039063	559.039063	2.5222842578	556.516778742
695.155877	695.155877	0.2325662578	694.923310742
5120.56897	5120.56897	20.568976346	5099.99999365
5120.56897	5120.56897	111.69856374	5008.87040626
10240.8596	10240.8596	1256.9632875	8983.89631255

Table II: Optimized and Non-optimized Query comparison in terms of efficiency

Number Of Rows	**Start Row No.**	**End Row No.**	**Execution time (m-sec) of Non Optimized Query**	**Execution time (m-sec) of Optimized Query**	**Difference (m-Sec)**
13696	70	80	210	150	60
54784	70	80	370	210	160
109568	70	80	570	250	320
219136	70	80	980	530	450
876544	70	80	9570	710	8860
3506176	70	80	18860	1230	17630

Table III: Comparison between traditional and optimized paging query techniques

Page Number	**Execution time (m-sec) using ADO**	**Execution time (m-sec) using Optimized solution**	**Difference in time (m-sec)**
10	71	73	2
100	127	137	10
500	650	310	340
10000	25806	2354	23452
100000	36452	3983	32469
200000	60213	5326	54887

Paging query process faces no problem in case of small data. If the amount of data increases then more client resources are required to cache the data which degrades the application performance. To overcome this problem, an optimized solution is needed. Since resources at DB server

are enough to process the query, we used key technologies in our proposed technique to improve the paging query efficiency. A major bottleneck for most of the large databases that affect paging query is the volume of data returned to the DB server from storage devices. The smart scan model is used to solve the problem of time spent to move unnecessary data from the storage tier to the database tier. The advantages of smart scan model in terms of efficiency and effectiveness over other traditional paging query techniques like ADO were observed.

5. CONCLUSION

The database pagination is sought by web developers and its optimization becomes necessary when dealing with large sized data. Paging query process requires more client resources to cache data. In classical database, the main problem is transferring huge data between storage systems and the DB servers. We presented a smart scan technique which is used to solve the problem of moving irrelevant data from storage tier to database tier. The presented solution for paging query tremendously improved efficiency and effectiveness of application as compared to the traditional paging query techniques.

References

A., H. and F., M. 2009. Evolution of Query Optimization Methods. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems I*, H. A., K. J., and W. R., Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 211–242.

Ali Khan, Z. and Khan, N. 2018. Data modeling and query optimization technique in business intelligence applications. *International Journal of Advanced Science and Technology 114*, 139–150.

Bach, M., Arao, K., Colvin, A., Hoogland, F., Osborne, K., Johnson, R., and Poder, T. 2015. *Expert Oracle Exadata*.

Bellamkonda, S., Ahmed, R., Witkowski, A., Amor, A., Zait, M., and Lin, C.-C. 2009. Enhanced subquery optimizations in oracle. *Proceedings of the VLDB Endowment 2,* 2, 1366–1377.

Bruno, N., Chaudhuri, S., and Ramamurthy, R. 2009. Power hints for query optimization. In *2009 IEEE 25th International Conference on Data Engineering*. 469–480.

Chaudhuri, S. 1998. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '98. ACM, New York, NY, USA, 34–43.

Eidson, W. C. and Collins, J. 2016. Methods and systems for joining indexes for query optimization in a multi-tenant database. US Patent 9,405,797.

Gupta, M. and Chandra, P. 2011. An empirical evaluation of like operator in oracle. *BVICAM's International Journal of Information Technology 3*.

Herodotou, H., Borisov, N., and Babu, S. 2011. Query optimization techniques for partitioned tables. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. ACM, New York, NY, USA, 49–60.

Ioannidis, Y. E. 1996. Query optimization. *ACM Comput. Surv. 28,* 1 (Mar.), 121–123.

Khan, M. and Khan, M. 2013. Exploring query optimization techniques in relational databases. *International Journal of Database Theory and Application (IJDTA) 6*, 11–20.

Li, D., Han, L., and Ding, Y. 2010. Sql query optimization methods of relational database system. *Computer Engineering and Applications, International Conference on 1*, 557–560.

Lohman, G. 2014. Is query optimization a solved problem. In *Proc. Workshop on Database Query Optimization*. Oregon Graduate Center Comp. Sci. Tech. Rep, 13.

O'Neil, P. and Graefe, G. 1995. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record 24,* 3, 8–11.

Sun, F. and Wang, L. 2011. Paging query optimization of massive data in oracle 10g database. In *2011 International Conference on Computer Science and Service System (CSSS)*. 2388–2391.

Sun, P., Zhao, Z., and Ge, Z. 2009. The research on the query optimization strategy of the vdsi system database. In *2009 Asia-Pacific Conference on Computational Intelligence and Industrial Applications (PACIIA)*. Vol. 1. 79–82.

Zafarani, E., Feizi Derakhshi, M., Asil, H., and Asil, A. 2010. Presenting a new method for optimizing join queries processing in heterogeneous distributed databases. In *2010 Third International Conference on Knowledge Discovery and Data Mining*. 379–382.

M.N.A. Khan obtained a D.Phil. degree in computer system engineering from the University of Sussex, UK. His research interests are in the fields of software engineering, cyber administration, digital forensic analysis and machine learning techniques.