The Effect of Horizontal Database Table Partitioning on Query Performance

Salam H. Matalqa & Suleiman H. Mustafa
Faculty of Information Technology and Computer Sciences
Department of Computer Information Systems
Yarmouk University, Irbid, Jordan
Salam.matalqah@yahoo.com & smustafa@yu.edu.jo

Abstract

The need for achieving optimal performance for database applications is a primary objective for database designers and a primary requirement for database end users. Partitioning is one of the techniques used by designers to improve the performance of database access. The purpose of this study was to investigate the effect of horizontal table partitioning on query response time using three partitioning strategies: zero partitioning, list partitioning and range partitioning. Three tables extracted from the Student Information System (SIS) at Yarmouk University in Jordan were used in this research. Variation in table size was used to determine when partitioning can have an impact (if any) on access performance. A set of 12 queries were run over a database of three different sizes. The results indicated that partitioning provided better response time than zero partitioning, on the other hand, range and list partitioning strategies showed little performance differences with the different database sizes.

Keywords:
Table Partitioning, Horizontal Partitioning, Range Partitioning, List Partitioning, Database Performance.

1. Introduction

Useful, accessible, and timely information has always been a great power for those who have it and use it efficiently. As such, gathering, managing, accessing and analyzing information have evolved to be a critical issue for the success of any kind of organization. With the rapid development of information technology, more and more large-scale application systems will generate vast amounts of data. Big data or massive data refers to the amount of data that cannot be captured, managed, processed, by the current mainstream software [20].

Based on the International Data Corporation (IDC) results, they show that the data produced in 2008, 2009, 2010 and 2011 by everyone is equal to more than 200GB. By the end of 2012, the amount of data rose from the TB (1024GB=1TB) level to PB (1024TB=1PB), EB(1024PB=1EB) and ZB (1024EB=1ZB) level [21]. By 2020, it is expected that the whole world generated data size will reach 44 times today. Consequently, big data tables will bring a great deal of performance pressures to application systems and a big risk in database management [19].
All information systems (ISs) such as telecommunication systems, banking systems, educational systems, health-care systems, and others depend on the management of data, and how to deal efficiently with the huge piles of data. Nowadays, we are living in an information era with tons of music, photos and videos. The task of data storing, sharing, organizing, and manipulating has become a challenge one. Hence, database management systems are considered the backbone and the heart of any application in our daily lives [14].
For any application that is already running in a production or for any new project that we are starting, performance is one of the most important aspects that should be taken into consideration. For database designers, achieving optimal performance is the primary objective, while for database end users it is a primary requirement. Developing and improving database performance is a cycling activity that should be included in each development stage. However, no recipe exists for designing perfect databases, but some techniques and tips can improve the quality of the design, such as indexing techniques and query optimization [10].

One of the most important aspects of physical database design is table partitioning which has significant impact on database performance and manageability of data. Partitioning subdivides a database object (table, an index

or an index-organized table) into smaller pieces. Each piece of the database object is called a partition which has its own name, and may optionally have its own storage characteristics. We divide database objects using a partitioning key, which is a set of columns that determine in which partition a given row will be located or stored.

According to [3], the three major benefits acquired from partitioning are the high performance (fast query response time), manageability (divide and conquer approach) and availability (independency of partitions). Furthermore backup and recovery operations can be done more efficiently and effectively with partitioning.

There are three strategies for partitioning tables or entities: horizontal, vertical or mixed (hybrid). Horizontal partitioning allows access methods such as tables, indexes and materialized views to be partitioned into disjoint sets of rows that are physically stored and accessed separately. It affects performance as well as manageability. On the other hand, vertical partitioning allows a table to be partitioned into disjoint sets of columns, and since many queries access only a small subset of columns in a table, vertical partitioning can reduce the amount of data that needs to be scanned to answer the query [1].

Mixed or hybrid partitioning is a combination of both types of partitioning, in which the table is divided into arbitrary blocks based on the needed requirements. It consists of horizontal partitioning followed by a vertical or a vertical partitioning followed by horizontal, when the schema is not be sufficient to satisfy the requirements by only one of them [6]. It is the most complex strategy and needs more management.

Horizontal partitioning is the most commonly used approach. Oracle offers three fundamental data distribution methods: range, list and hash. Range partitioning is the most common type of horizontal partitioning, which maps data into partitions based on ranges of values of the partitioning key that we select for each table.

In comparison, list partitioning is based on specifying a list of discrete values for the partitioning key that enables us to explicitly control how rows map to partitions. It has an advantage in that we can group and organize unordered and unrelated sets of data in a natural way.

Finally, hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that we identify. Each partitioning strategy has different advantages and design consideration, such that each strategy is more appropriate for a particular situation. Figure 1, shows a simple representation of the three horizontal partitioning techniques [4].

These three techniques are usually described as one-level partitioning approach. On the other hand, the three techniques can be combined in different ways in what known as composite (multi-level) partitioning. Combinations include Range-range, range-hash, range-list, list-range, and others.

There are some suggestions or situations for when it is more suitable to partition a table. A general advice is to partition when table size is greater than 2 GB. A candidate situation for partitioning is when a table contains historical data so that the new data is added into the newest partitions [4].

The powerful functionality of Oracle partitioning solves the problems and negative impacts of big data tables. It is driven by and depends on business requirements. However, Oracle somehow does not provide clear differentiation between query response time measures, since it needs very huge data set to see the differentiation. As such, Microsoft SQL Server platform has been used in this study.

This paper presents the results of investigating the effect of horizontal partitioning on query performance. Two strategies of partitioning (namely, range and list partitioning) have been used and their performance has been compared with no partitioning. It is organized as follows: section two presents some related works, section three presents the methodology used in the study, section four discusses and evaluates the results, and finally section five is devoted for the conclusion.

2. Related Work

Round-robin partitioning (in which every tuple inserted will be assigned to a different fragment or partition so that rows will be distributed evenly and in a ring fashion), hash partitioning, and range partitioning are the most popular horizontal approaches used [12]. In general, hash based partitions are good for clustering only when the queries contain equality predicates on the partitioning attributes. On the other hand, BigTable presented by [9] and PNUTS presented by [11] use key-based range partitioning.

Horizontal partitioning or fragmentation using min-term predicates was first introduced by [7] for distributed databases. They considered the problem of horizontally partitioning data on a set of resources. A methodology was proposed for determining the access parameters that are performed over different portions of data, and the concepts required for the determination of the relevant one were identified. The general partitioning problem was formulated in three specific application environments, showing that the solution models require exactly the concepts and parameters introduced.

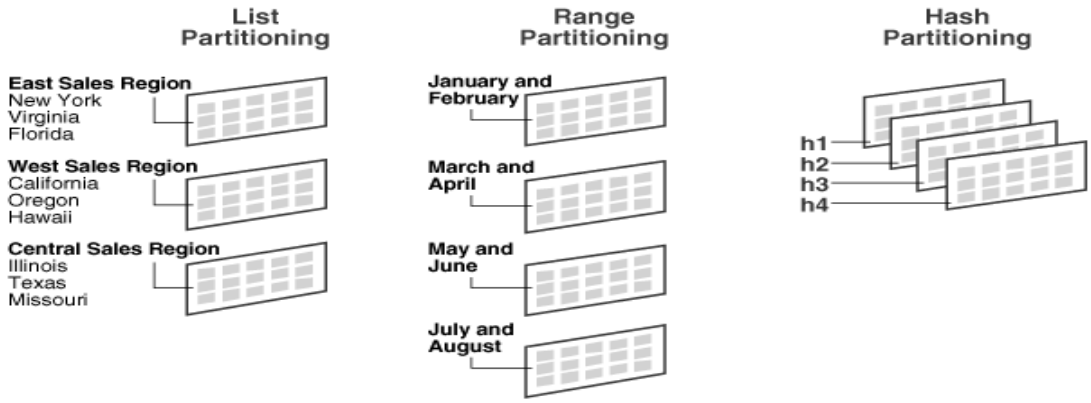

Figure 1: Horizontal partitioning techniques: List, Range and Hash [22]

Cheng et al. presented a fragmentation approach in [9] based on a genetic algorithm (GA) to achieve high database retrieval performance, by treating horizontal fragmentation as a travelling salesman problem (TSP). They also proposed three new operators for GAs. The experimental results indicated that these operators outperformed other operators in solving the TSP. The data partitioning problem was solved by applying this proposed GA, and the computational study showed that their GA outperforms well for this application.

Ma et al. presented a heuristic approach in [16] for using derived horizontal fragmentation, which depends on a cost model for analyzing the cost of queries. They wanted to provide a tractable approach to minimize the query processing costs by performing horizontal fragmentation and fragment allocation simultaneously. Some experiments were conducted to verify their algorithm. The results showed that this heuristic approach outperformed the traditional approaches in terms of system performance. But, they observed that the processing time spent in testing their approach was similar to that spent in using the traditional approach. The improvement of performance was not significant, but for some other database instances and queries, they expected better performance improvements.

In the previous research studies, type and frequency of queries were important for applying partitioning solutions. However, for a distributed system, these solutions are not suitable at the initial stage of a database design. Khan and Hoque have presented a fragmentation technique in [15] for partitioning tables that can be applied at the initial stage as well as in later stages of a distributed database system. This technique depends on the use of Attribute Locality Precedence (ALP) which means fragmenting a relation horizontally based on locality of precedence of its attributes. ALP represents the value of importance of an attribute with respect to sites of distributed database. Database designer is responsible for constructing an ALP table for each relation of a DDBMS during database design stage. CRUD (Create, Read, Update, and Delete) matrix and cost functions are used in combination with the ALP table. Results showed that for relational databases in distributed systems, this proposed technique can solve initial fragmentation problems properly.

In the design phase of distributed databases, improving performance is an important aspect to take into consideration. Horizontal partitioning has an important impact in achieving this performance need. Distributed databases are becoming very popular nowadays. In the view of [6], making proper fragmentation for relations and allocating fragments are not easy tasks. Many techniques have been proposed by the researchers, such as using empirical knowledge of data access and query frequencies, but doing proper fragmentation and allocation at the initial stage of a distributed database has not yet been addressed. Bhuyar et al. have proposed a fragmentation technique in [6] to partition relations properly at the initial stage for distributed databases when no data access statistics and query execution

frequencies are available. Their Results were similar to those of [15]. They demonstrated that the proposed technique can solve initial fragmentation problem of relational databases for distributed systems properly.

Ezeife and Barker reviewed the taxonomy of class models in [13] for the fragmentation problem in the distributed object database, and presented a comprehensive set of algorithms for horizontally fragmenting this taxonomy of class models. Their approach starts with generating primary horizontal fragments for a class, based on only applications that access this class. Next it generates derived horizontal fragments that arise from primary fragments of its subclasses, such as its complex attributes (contained classes), and/or its complex methods classes. Based on the queries accessing the class, primary horizontal partitioning was performed using predicates of these queries.

Derived horizontal partitioning for a class was based on the horizontal partitioning of another class. The sets of primary and derived fragments of each class were combined to produce the best possible fragmentation scheme. Thus, their algorithms support inheritance and class composition hierarchies, as well as nesting methods among objects. They have been shown to have polynomial time.

Bellatreche et al. presented some algorithms in [5] for both primary and derived horizontal partitioning. They discussed the problems of localization of fragments for queries, and the migration of objects for updates. These two issues are important aspects for supporting database operations on a partitioned database. For a given query, the horizontal fragments that result from this query can be identified easily with fragment localization, and if we need to migrate an object form one fragment to another due to updates, we deal with object migration issues. Finally they showed the benefits of horizontal partitioning for query processing.

For object oriented distributed database systems (OODD), Areed et al. proposed a new algorithm in [2] for applying horizontal partitioning over these systems. They applied both horizontal and vertical ideas for relational systems, such that in the context of horizontal partitioning of an object model, they identified vertical partitioning and allocation simultaneously. They used a cost model to minimize the global fragmentation and allocation costs, and used simulation to validate the proposed approach. Compared to most recent affinity-based horizontal partitioning, the study proved that the proposed approach was simpler and had less cost.

Silva et al. studied and proposed guidelines in [18] to be used in XML databases when applying a fragmentation design algorithm, with the aim of increasing query processing performance. They used broader aspects that could be further considered during the fragmentation design. Experiments were performed over different sizes of XML databases to assess how data growth impacts the performance of query processing. Their experiments showed that there are performance gains obtained from the fragmentation process for frequent queries, compared to the results obtained in the centralized environment. They obtained gains even for queries that did not apply over fragments. A set of recommendations were suggested for choosing the best type of horizontal fragmentation to be applied to a particular XML databases.

Mahboubi and Darmont worked on XML warehouse fragmentation in [17]. They proposed the use of derived horizontal fragmentation over XML contexts. They also compared the two primary horizontal fragmentation methods: predicate construction and affinity-based fragmentation. Their experiments confirmed that derived horizontal fragmentation improved query response time significantly, and in all their experiments, the affinity-based fragmentation clearly outperformed predicate construction. They claimed that this had never been demonstrated before as far as they know, even in the relational context.

3. Methodology

In this study, a database consisting of three tables was used to perform and evaluate partitioning strategies. The three tables represent part of data about courses and course sections offered at Yarmouk University in Jordan. A set of SELECT queries were used to determine which of the two strategies would achieve better performance. Queries were run over these tables, once without table partitioning and once with table partitioning.

For each partitioning strategy, queries were run three times over different sizes of tables. By doing so, we aimed to explore the effect of table partitioning strategies and table size on query response time.

The dataset used in this study was extracted from the student information system (SIS) at Yarmouk University (YU) [21]. Only three tables were selected for this purpose: the first table contained information about courses registered by the students at the College of Information Technology, the second table stored information about their degree plans and the third table was used for course sections.

The data was exported to an Excel sheet then imported to three databases in MS SQL server platform. Figure 2 presents a partial conceptual schema for these three tables. The tables representing this schema are as follows:

T1*: STUDENT (S_ID, F_Name, L_Name, B_Date)*
T2*: CORSE (C_ID, C_Name, Credits)*
T3*: TAKES (S_ID, C_ID, Taken, Prerequisite)*
T4*: COURSE_SECTION (Sec_ID, Sec_No, Room, Room_Size, Instructor, Sec_Days, Sec_Time, C_ID)*

Initially, each table had about one thousand records. Then, to show the effect of various partitioning strategies, the size was increased twice for the two tables to be partitioned: TAKES and COURSE_SECTION. As such, three database versions were implemented: The first version included about one thousand records for each of these two tables, the second version included about four thousand records for each, and the third contained about nineteen thousand records.

The courses table was not partitioned, since it contained no suitable candidate partitioning keys for Range partitioning. For TAKES, the student ID attribute (**S_ID)** was used as a range partitioning key, and the attribute **Taken** was used as a list partitioning key.

Finally for COURSE_SECTION, the section time attribute (**Sec_Time**) was used as a range partitioning key, and the section days attribute (**Sec_Days**) as a list partitioning key. Table 1 and Table 2 show the distribution of records for each partitioning strategy using three different table size versions.

The set of queries used in this study are listed in Appendix I. As Table 3 shows, most queries require inner joins between two tables or more. They were designed to retrieve records based on conditions that combine the partitioning attribute keys. The purpose was to show which partitioning strategy would provide better performance for each query in terms of response time.

Some queries, like the third and the sixth, retrieve records based on conditions that combine partitioning attribute keys form multiple tables (Inner JOIN conditions). This was intended to show if Range partitioning strategy or List partitioning strategy would be better for each of the two relations: *TAKES* and *COURSE_SECTION*.

The twelve SELECT queries were executed over the three database versions with different sizes, using the same table structures and attribute partitioning keys. Each execution covered the three partitioning strategies: No partitioning, Range partitioning, and List partitioning. As shown in Table 1 and Table 2, the difference between these three experiments was only in the database size. In the first the database contained about three thousand records, while in the second and third executions the size was increased for tables: TAKES and COURSE_SECTION.

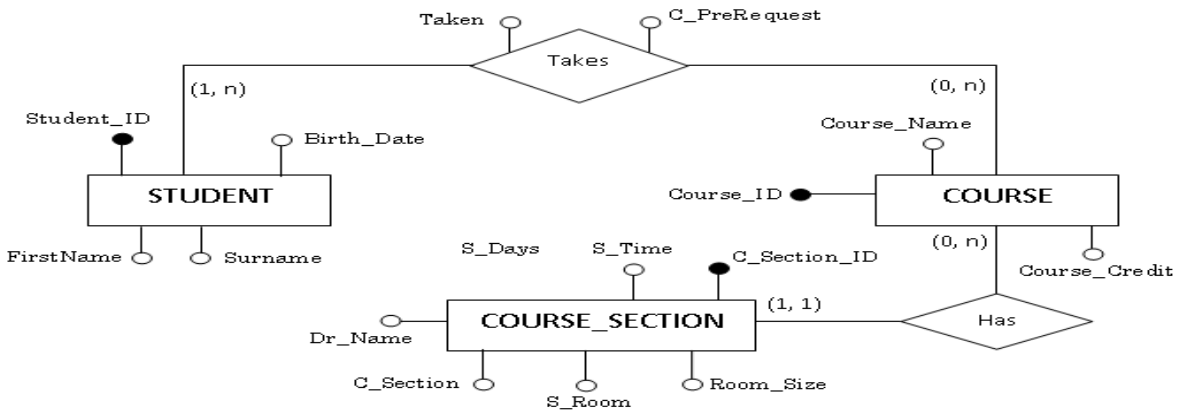

Figure 2: Partial ER diagram extracted from the student registration subsystem at YU

Table 1: No of records contained in each partition for the table: **TAKES**

Table Size Version	# records (**No partitioning)**	# records (**Range** Partitioning (S-ID))		# records **(List** Partitioning (Taken))	
		Partition 1	Partition 2	Partition 1	Partition 2
Size 1	1192	609	583	328	864
Size 2	4768	2436	2332	1312	3456
Size 3	19072	9744	9328	5248	13824

Table 2: No of records contained in each partition for the table: **COURSE_SECTION**

Table Size Version	# records (**No Partitioning)**	# records (**Range** partitioning (S-Time))		# records (**List** partitioning (S_Days))	
		Partition 1	Partition 2	Partition 1	Partition 2
Size 1	1130	655	475	647	475
Size 2	4520	2620	1900	2588	1900
Size 3	18080	10480	7600	10352	7600

Average response time (ART), as defined bellow, was used as measure of performance for comparing the three partitioning strategies. Response time has been defined as the elapsed time in milliseconds from the moment that a query is entered at the interface to the time that the application indicates the query has completed and results shown.

$$ART = \left(\sum_{i=1}^{n} RT \right) / n$$

Where, ART: Average response time, *RT*: Response time for each query., *i*: Query number, and *n*: Number of queries.

3. Results and Evaluation

Figure 3 presents the results of executing the queries using a database of about three thousand records. The results indicate that partitioning exhibits better performance in terms of response time than no partitioning, with about 18-22% improvement. However, range partitioning and list partitioning provided almost similar results. This might be attributed to the relatively small size of database tables used in this query execution round.

In comparison, when the size of tables being partitioned was scaled up to more than four thousand records, we could notice some difference in performance between range partitioning and list partitioning. As Figure 4 show, range partitioning outperformed list partitioning in the average response time with about 18% difference.
As in the previous case, both partitioning strategies provided better response time performance than no partitioning. Improvement realized was about 15-30%.

When the database was scaled up to about eighteen thousand records for each table partitioned, the results, as exhibited in Figure 6, showed no real difference between range partitioning and list partitioning. What is also more notable is that the difference in performance between no partitioning and partitioning is relatively small. As Figure 5 shows, the average response time of partitioning provides only about 15-18% improvement over no partitioning.

There are no other results from previous research to compare with. One might assume that range partitioning and list partitioning behave similarly in performance in view of the kind of database tables used and number of queries used regardless of the database size.

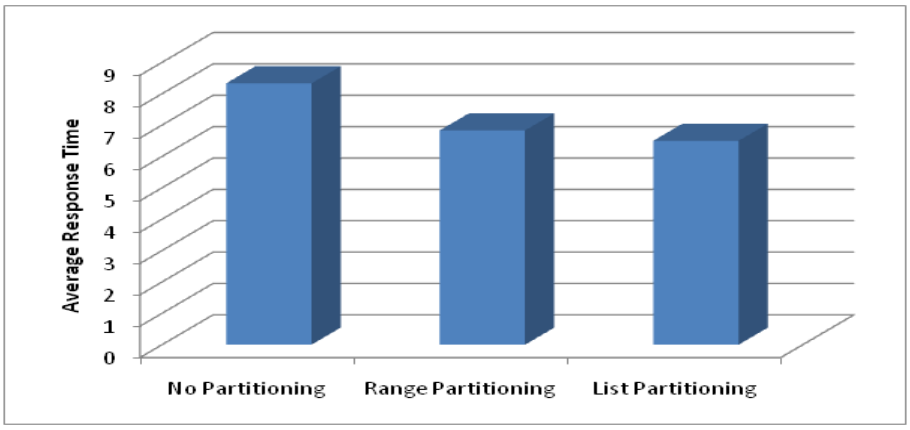

Figure 3: Average response time for the three strategies for the first database size (Size-1)

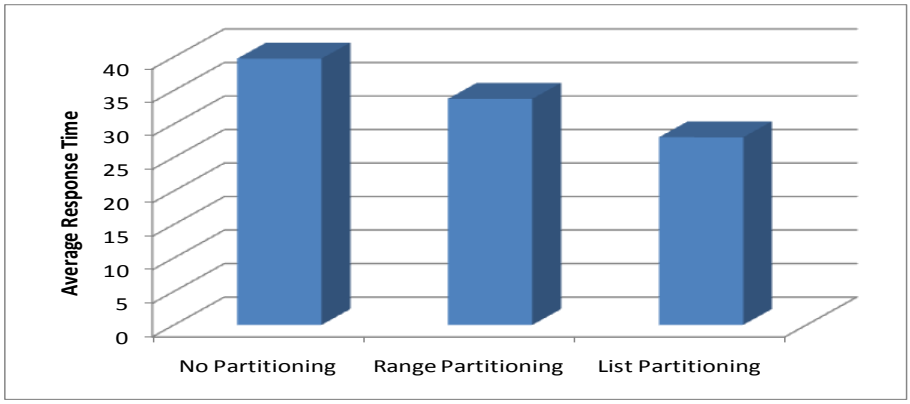

Figure 4: Average response time for the three strategies for the second database size (Size-2)

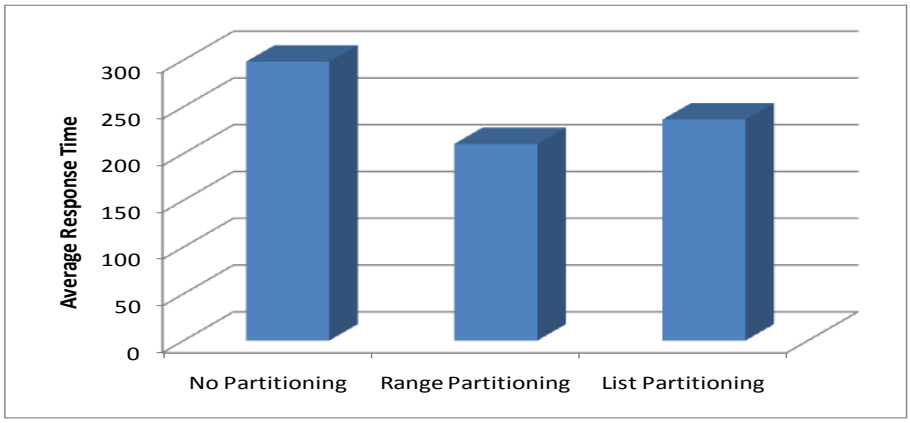

Figure 5: Average response time for the three strategies for the third database size (Size-3)

4. Conclusion

Many factors can affect partitioning decisions that would be taken over database tables, such as size of the database, type of data, type of queries, frequency of queries, partitioning attribute keys, etc. The results reported in this study should be viewed within the kind of data and tables used, the kind and number of queries used, and the type of partitioning strategy investigated. The tables used in this study have been extracted from

a database, which means that that we are not dealing with a full database environment in a real setting.

Generally the results confirm that partitioning improves query response time over non-partitioning. Nevertheless, one still should ask how much improvement is acceptable in view of the overhead cost which results from partitioning. Given the size of the tables used, the results do not show significant improvement with partitioning. The general implication of this is that partitioning should be applied for only when we have reasonably large data tables. Moreover, this study considers only select queries. If we consider update operations, would an improvement of some level in performance still be realized? Such question is important in deciding to go for partitioning.

In comparing Range partitioning strategy with List partitioning strategy, no real difference was shown in the results of the study. There is no absolute ultimate choice or decision for table partitioning for any database, in terms of type of partitioning and the selection of partitioning keys for each table. Each strategy can be useful in specific situations. Range and List are not comparable for the same partitioning keys in a certain table, because each is useful and suitable for specific type of attributes. It might be useful for further research on this issue to consider a larger number of tables and queries.

References

[1] Agrawal S., Narasayya, V. and Yang, B., "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design". *SIGMOD, ACM*, Paris, France, pp 359-370, 2004.

[2] Areed M., El-Dosouki, A. and Ali, H., "A heuristic approach for horizontal fragmentation and alllocation in DOODB", *In Proc. INFOS2008*, Cairo, Egypt, pp. 9-16, 2008.

[3] Baer H., *Partitioning in Oracle Database 11g*, Oracle, USA, 2007.

[4] Baer H. et al., *VLDB and Partitioning Guide, 11g Release 2 (11.2)*, Oracle, USA, 2010.

[5] Bellatreche L., Karlapalem, K. and Simonet A., "Algorithms and support for horizontal class partitioning in object-oriented databases", *Distributed and Parallel Databases*. vol. 8, no. 2, pp.155–179, 2000.

[6] Bhuyar P., Gawande, A. and Deshmukh, A., "Horizontal Fragmentation Technique in Distributed Database", *International Journal of Scientific and Research Publications*, vol. 2, no. 5, pp 1-7, 2012.

[7] Ceri S., Negri, M. and Pelagatti, G., "Horizontal data partitioning in database design", *In Proc. ACM SIGMOD*, Milano, Italy, pp. 128–136, 1982.

[8] Chang F. et al., "Bigtable: a distributed storage system for structured data", *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp 1–26, 2008.

[9] Cheng C, Lee, W. and Wong, K., "A genetic algorithm-based clustering approach for database partitioning", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 32, no. 3, pp. 215–230, 2002.

[10] Cioloca C. and Georgescu M., "Increasing Database Performance using Indexes", *Database Systems Journal*, vol. 2, no. 2, pp 13-22, 2011.

[11] Cooper B. et al., "PNUTS: Yahoo!'s hosted data serving platform", *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp1277–1288, 2008.

[12] DeWitt D and Gray J., "Parallel database systems: the future of high performance database systems", *Communications of the ACM*, vol. 35, no. 6, pp 85–98, 1992.

[13] Ezeife C. and Barker K., "A comprehensive approach to horizontal class fragmentation in a distributed object based system", *Distributed and Parallel Databases*, vol. 3, no. 3, pp 247–272, 1995.

[14] Idreos S., "Database Cracking: Towards Auto tunning Database Kernels". *Aan de Universiteit van Amsterdam*, geboren te Lesvos, Griekenland, 2010 (Thesis).

[15] Khan S. and Hoque A., "A New Technique for Database Fragmentation in Distributed Systems", *International Journal of Computer Applications*, vol. 5, no. 9, pp 20 – 24, 2010.

[16] Ma H., Schewe, K. and Wang, Q., "A Heuristic Approach to Cost-Efficient Derived Horizontal Fragmentation of Complex Value Databases", *In Proc. Eighteenth Australasian Database Conference (ADC 2007)*, Ballarat, Australia, pp 103-111, 2007.

[17] Mahboubi H. and Darmont J., "Enhancing XML Data Warehouse Query Performance by Fragmentation", *Proceedings of the 2009 ACM symposium on Applied Computing*, New York, USA, pp 1555-1562, 2009.

[18] Silva T. et al., "Towards Recommendations for Horizontal XML Fragmentation", *Journal of Information and Data Management*, vol. 4, no. 1, pp 27–36, 2013.

[19] Singh S. and Singh N., "Big Data Analytics[C]", *International Con. on Communication,*

Information & Computing Technology (ICCICT), Mumbai, India, pp 1-4, 2012.

[20] Zhu M. and Zhang X., "The Management of Big Data Tables Based on Oracle Partition Technology", *The 9th International Conference on Computer Science & Education*, Vancouver, Canada, pp 570-572, 2014.

[21] *Yarmouk University 1976*, Course Schedule of the Faculty of Information Techonolgy and Comp. Sciences. Retrieved December,9, 2014 from: http://admreg.yu.edu.jo/index.php?option=com_content&view=article&id=253&Itemid=438

[22]http://docs.oracle.com/cd/E11882_01/serve.112/e25523/partition.htm

Salam H. Matalqa obtained her M.S degree in Computer Information Systems (CIS) in June 2015, from Yarmouk University, Irbid, Jordan, and her B.S. degree in Computer Information Systems in January 2009, from the same university. Her research interests tend to focus on areas of database systems, information retrieval and data mining.

Suleiman Hussein Mustafa is a professor of Information Systems and is currently the Dean of the Faculty of Information Technology & Computer Sciences at Yarmouk University since 2/9/2012. He got his Ph.D. from the University of Pittsburgh (USA) in 1986. He worked in several universities and was assigned several academic and management positions. He has published more than thirty papers in a number of research areas in computer science and information systems including natural language processing, database and information retrieval systems, and software engineering.

Appendix I

List of Queries Used in the Study

1- Retrieve course IDs and course names for all courses that were taken by students whose numbers are greater than 2010901150.
2- Retrieve course IDs, course names, and prerequisites for all courses that are still not taken by students, whose numbers are less than or equal 2010901150.
3- Retrieve course IDs, student IDs and instructors for all students whose IDs are less than 2010901100 and their course sections start at 8:00.
4- Retrieve course IDs and course names for all courses that were taken by students whose numbers are greater than 2010901150.
5- Retrieve course IDs, section IDs and instructors for all course sections that start at 8:00 on the days of Sunday, Tuesday or Thursday (separately).
6- Retrieve course IDs, section IDs and rooms for all courses that were taken by students whose course sections held on Sunday, Tuesday and Thursday (together) at 14:00 and onward (after that).
7- Retrieve course IDs and course names for all courses that were taken by students, whose IDs are less than or equal 2010901150.
8- Retrieve course IDs, course names, and course prerequisites for all courses that were still not taken by students whose IDs are greater than 2010901150.
9- Retrieve course IDs, student IDs, and instructor names for all students whose IDs are greater than 2010901175 and their taken-course sections start at at 14:00 and onward (after that).
10- Retrieve course IDs, sections and instructors for all course sections that were held between 12:00 to 14:00, on the days of Sunday, Tuesday or Thursday (together).
11- Retrieve course IDs, sections, and instructors for all course sections that start at 8:00 on the days of Monday and Wednesday (separately).
12- Retrieve course IDs, sections, and rooms for all courses that were taken by students whose course sections held on Monday and Wednesday (together) at 9:00 and before.