Aging Locality Awareness in Cost Estimation for Database Query Optimization

Chihiro Kato[1](✉), Yuto Hayamizu[1], Kazuo Goda[1], and Masaru Kitsuregawa[1,2]

[1] The University of Tokyo, Komaba 4-6-1, Meguro-ku, Tokyo, Japan
kato@tkl.iis.u-tokyo.ac.jp
http://www.u-tokyo.ac.jp/

[2] National Institute of Informatics, Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo, Japan
http://www.nii.ac.jp/

Abstract. A number of insertions, updates and deletions eventually deteriorate the structural efficiency of database storage, and then cause performance degradation. This phenomenon is called "aging." In real-world database systems, aging often exhibits strong locality because of the inherent skewness of data access; specifically speaking, the cost of I/O operations is not uniform throughout the storage space. Potentially query execution cost is influenced by the aging. However, conventional query optimizers do not consider the aging locality; thus they cannot accurately estimate the cost of query execution plans at times. In this paper, we propose a novel method of cost estimation that has the key capability of accurately determining aging phenomena, even though such phenomena are non-uniformly incurred. Our experiment on PostgreSQL and TPC-H data sets showed that the proposed method can accurately estimate the query execution cost even if it is influenced by the aging.

Keywords: Database systems · Query optimizer · Database aging

1 Introduction

The structural efficiency of database storage is fundamental to the query execution performance in database systems. Insertions, updates and deletions can scatter densely packed records and disturb the physical ordering of records. Repeated execution of these operations eventually deteriorates the structural efficiency and then causes performance degradation. This phenomenon is called "aging". Aging can greatly affect the I/O cost of query execution, and its influence is different for each candidate plan owing to the difference of I/O strategies. The progress of aging phenomenon can cause errors in cost estimation. Thus query optimizers may choose non-optimal plans because they are unaware of aging phenomenon.

C. Kato—Currently, Fujitsu Laboratories.

The original version of this chapter was revised: The authors' affiliations were incorrect. This has been corrected. An erratum to this chapter can be found at 10.1007/978-3-319-44406-2_40

S. Hartmann and H. Ma (Eds.): DEXA 2016, Part II, LNCS 9828, pp. 389–396, 2016.
DOI: 10.1007/978-3-319-44406-2_32

The difficulty of aging-aware cost estimation is that aging often has strong locality due to the skewness of data access by user activities. Actual cost of query executions can differ even when queries are the same except for access ranges. While the significance of aging on database performance has been recognized from the early history of database systems, awareness of aging locality in query optimization has been remained largely unexplored to the best of our knowledge.

In this paper, we propose a novel method of cost estimation for query optimization that has the key capability of figuring out the aging phenomenon accurately even though this occurs non-uniformly. Our experiments showed that the proposed method yields good cost estimation and helps the choice of optimal query plans. The rest of this paper is organized as follows. We describe the proposed method in Sect. 2. We then present the evaluation of the proposed method in Sect. 3. We summarize related work in Sect. 4 and conclude the paper in Sect. 5.

2 Aging Locality Aware Cost Estimation

2.1 Influence of Aging and Its Locality on Query Optimization

While initially loaded databases can enjoy good efficiency, repeated execution of insertions, updates and deletions eventually disturb the physical ordering of a table, spatially scatter records across a table, and then degrade performance. This phenomenon is called aging. As described in the previous section, databases in production are inherently in aged states for most of their lifetimes.

In terms of query optimization, this performance degradation due to aging means the increase of the I/O cost. This cost increase has two aspects: temporal and spatial variation. Both can lead to wrong choices of query execution plans in different ways. The temporal variation of the I/O cost is caused by the progress of aging and can change the optimal query execution plan for a certain query. The spatial variation of the I/O cost is caused by aging locality. Even if queries are the same except for access ranges, as seen in prepared statements, the optimal query execution plans can be different in the presence of spatial variation of the I/O cost.

In situations with aging locality, a conventional optimizer cannot reflect the spatial variation of the I/O cost in the cost estimation, which can result in choosing non-optimal query execution plans. In order to choose the optimal query execution plan on aged databases, cost models should be aware of aging locality. In the next subsection, we present I/O cost models with aging locality and provide a method to measure the increase of the I/O cost for the presented models.

2.2 I/O Cost Models with Aging Locality

First, we model the I/O access cost for only one table. We define $S(x)$ as a window function of the access range, and $D(x)$ as the distribution density of data, where x can be the value of an indexing key or an address in a table space. If the cost $C(x)$ of accessing a record pointed by x is given, the I/O cost for a single table access can be described as follows:

$$\Gamma = \int S(x)D(x)C(x)dx \tag{1}$$

$S(x)$ is equal to 1 if x is in an accessed range; otherwise, it is 0. $D(x)$ denotes the number of records for x. When the table is initialized or reorganized, $C(x)$ should be nearly a constant; as the table ages, $C(x)$ changes its shape according to the increase of the I/O cost. Note that we do not consider a composite primary key in this paper; this will be left for future work.

For join queries, we combine these functions to estimate the I/O cost. For example, a nested loop join query picks up matching records in table t_1 one by one. For each record in table t_1, scans records in table t_2 that satisfy join conditions. Thus, its I/O cost can be described with the join cardinality $j_{t12}(x)$ between tables t_1 and t_2 as follows:

$$\Gamma_{NLJ} = \int S_{t1}(x)D_{t1}(x)C_{t1}(x)dx + \int j_{t12}(x)\{S_{t1}(x)D_{t1}(x)\}C_{t2}(x)dx \tag{2}$$

On the other hand, the I/O cost of hash join queries are rather simple:

$$\Gamma_{HJ} = \int S_{t1}(x)D_{t1}(x)C_{t1}(x)dx + \int S_{t2}(x)D_{t2}(x)C_{t2}(x)dx \tag{3}$$

In order to calculate a value of the I/O cost of a requested query, $C(x)$ must be available before query requests. In this paper, we focus on two fundamental access methods; full-table scan and index scan. We employed a measurement-based approach with performance test queries to approximate $C(x)$ for each access method. For the full-table scan, regardless of the actual $C(x)$, the average of the I/O cost increase is enough for cost estimation, so its performance test query is just a simple full-table scan. For the index scan, the x-space is divided equally into N parts, and performance test queries are given as index scan queries of each part. By measuring the execution time of these performance test queries beforehand, approximate values of $C(x)$ can be provided for our cost models.

The purpose of this paper is showing that aging locality aware cost modeling can improve the accuracy of cost estimation. This approach requires non-negligible amount of workload. We would like to further investigate efficient calculation of $C(x)$ in future.

3 Experiment

In order to validate the potential benefits of the proposed cost estimation, we performed intensive experiments by using an open-source database system and an industry-standard benchmark data set.

Table 1. Experimental setup

Server model	Dell PowerEdge R720xd
Processor	2x Intel Xeon E5-2690 v2
Main memory	64 GB DRAM
Storage devices	1x 900 GB HDD dedicate for database
	1x 900 GB HDD dedicate for operating system
Operating system	CentOS release 5.8 (64 bit)
Database system	PostgreSQL 9.4.0 (buffer size 128 MB)
Data set and schema	TPC-H, dbgen 2.17.0

3.1 Experimental Setup

Table 1 summarizes the laboratory environment that we built. PostgreSQL was configured with default configuration parameters unless specially noted.

First, we generated an initial data set by executing dbgen with a scale factor 100 and loaded the data set into the PostgreSQL database. After loading the data set, we executed the VACUUM command because this is well-known as a best practice to obtain the maximum performance. Following this, we performed a measurement; we executed a query and measured execution information, such as the taken execution time and deployed query execution plan. Note that, every time we started execution of a query, we cleaned up Linux disk buffer and PostgreSQL database buffer to measure cold-start performance by preventing some data from being cached there.

After we completed a measurement in the initial status, we iterated a bulk update on the database and took another measurement on the updated database. We performed the bulk update by executing refresh functions generated by dbgen. Logically, the database size does not change even as we update the database. To ensure fair measurement, we also ran the VACUUM command every time we completed a refresh function. By iterating database refreshing and performance measurement, we observed how query execution behavior would change as we incrementally updated the database.

```
SELECT SUM(l_extendedprice) FROM lineitem
WHERE l_orderkey < x AND l_orderkey > y                          ··· (1)

SELECT SUM(l_extendedprice) FROM lineitem
WHERE l_partkey < x AND l_partkey > y                            ··· (2)
```

Fig. 1. Test queries (Example)

```
SELECT SUM(l_extendedprice) FROM lineitem, part
WHERE p_partkey < x AND l_orderkey > a AND l_orderkey < b
AND l_partkey = p_partkey                                             ···(A)
```

Fig. 2. Validation queries

3.2 Cost Estimation Accuracy

This section presents the experimental results that validate the benefits of the proposed cost estimation.

We performed a measurement for the initial (non-aged) status and the refreshed (aged) status in the same database for each table and each access method. The refreshed status meant that the refresh function (updating the 10 % of the storage space) had been performed four times. For each measurement, we first performed each test query (example is depicted in Fig. 1) to measure aging degrees throughout the database. For example, regarding the test queries (1)–(2), we performed the query for different combinations of x and y so that the series of query executions would eventually cover the whole database space. Specifically, we divided the key space described by l_orderkey into ten pieces. In the first query trial for the test query (1), we set (x, y) to (min(l_orderkey), min(l_orderkey) $\times$ 9/10 + max(l_orderkey) $\times$ 1/10). As well, in the second query trial, we set (x, y) to (min(l_orderkey) $\times$ 9/10 + max(l_orderkey) $\times$ 1/10), min(l_orderkey) $\times$ 8/10 + max(l_orderkey) $\times$ 2/10). And we execute the same query with different (x, y) until we could cover the whole key space to obtain aging degrees over the space.

Based on the measured aging degrees, we estimated the query cost for the validation query (depicted in Fig. 2) in accordance with the estimation method introduced in Sect. 2. We also actually performed the validation query and compared the estimated cost and actual execution time to investigate how accurately the proposed method could estimate the query execution costs.

For comparison, we also measured the estimated cost reported by the EXPLAIN command in PostgreSQL to execute the validation query. This estimated cost is an internal value that is used for query optimization in PostgreSQL.

Figure 3(a) and (b) present aging degrees that we measured over the key space described by l_orderkey for the initial status and aged status, respectively, of the same database. As is clearly illustrated, access cost were uniformly distributed with the initial status, but in the aged status, access cost in the first 10 % region dramatically increased. In other words, aging phenomena were incurred in this region.

Figure 4(a) shows how query optimization was performed for query (A) with $x = 1000$ in the aged status. Aging was incurred in a limited portion in the database. To investigate the aging locality, we set $a = 60,000,000$ and $b = 120,000,000$ so that the query could fall in the non-aged region and we set $a = 0$ and $b = 60,000,000$ so that it could go into the aged region. The graph summarized the EXPLAIN cost and actual execution time for two different query

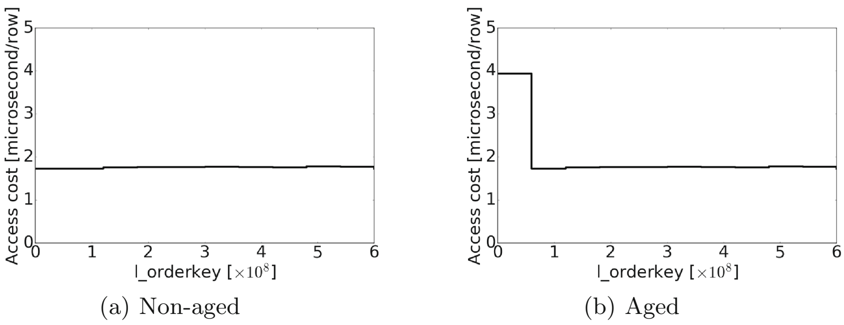

Fig. 3. Aging degrees of l_orderkey

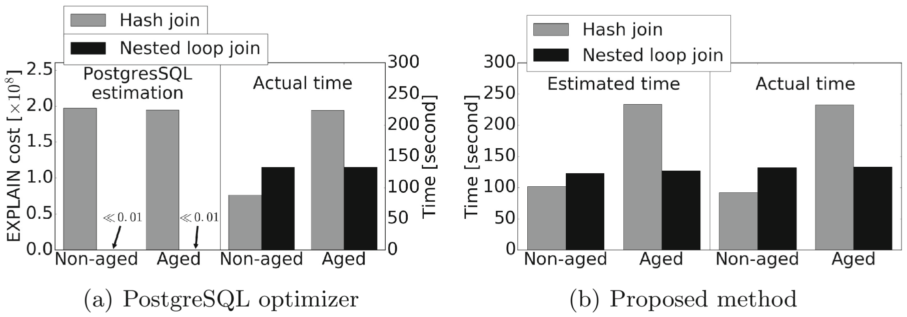

Fig. 4. Compare estimated cost with execution time about query (A) (selectivity of part table is 0.005 %)

execution plans; nested-loop join and hash join. In both regions, PostgreSQL estimated much smaller cost for nested loop join rather than hash join. In terms of the actual execution time, however, hash join outperformed nested loop join for the non-aged region but vice versa for the aged-region. This experiment confirmed that the current implementation in PostgreSQL cannot accurately estimate aging phenomena that were incurred in the database storage.

In contrast, Fig. 4(b) presents the estimated cost with the proposed method for the same query configuration. As is clearly shown, the proposed method successfully obtained a lower cost for hash join in the non-aged region but for nested loop join in the aged region.

4 Related Work

4.1 Aging and Database Reorganization

Performance degradation due to aging phenomenon has been a big headache for database administrators. Besides a mathematical analysis of performance degradation [1], database reorganization has been studied as a practical solution. In the ‘70s, the size of the database was generally small. Off-line reorganization was an reasonable approach [2], and arbitration between performance degradation and reorganization cost was the main concern at that time [3,4].

As the size of databases in operation grew rapidly, online reorganization became mainstream. Online reorganization technologies are largely placed into two categories: replicating a database and writing back the result afterwards [5], and incrementally reorganizing a database in place avoiding conflict with running queries by users [6]. Starting from the ‘80s, online reorganization has remained an active field of research [7,8] to the present.

However, despite intensive studies on database reorganization, these approaches still require too many resources to be executed frequently enough for keeping databases from being aged. In realistic situations, a certain level of aging phenomenon is unavoidable. In this paper, we propose a novel method of aging-aware cost estimation for query optimization. The proposed method can accurately estimate the cost of query execution even in the existence of non-uniform aging phenomenon, while conventional methods cannot.

4.2 Query Optimization

A query optimizer is a key component of database systems that converts an incoming query into an optimal query execution plan [9]. It has been studied intensively and extensively [10,11], such as parallelization of query optimization for utilizing the increasing number of CPU cores [12], caching results for future query optimization [13], and so on. In recent years, query optimization for emerging parallel query engine has been actively studied, such as using intermediate results for query optimization [14]. I/O cost modeling is a centerpiece of cost-based optimization. Storage systems were mostly based on magnetic disks before the 2000s [15], but in recent years Flash-based SSDs have increased its adoption rapidly in enterprise systems, and revisiting the I/O cost modeling has gained momentum [16]. In this paper, we focus on I/O cost modeling in the presence of non-uniform aging phenomenon.

5 Conclusion

We proposed a novel technology of query cost estimation that has the key capability of figuring out aging phenomena accurately even though the aging phenomena are non-uniformly incurred on the storage space. Our experiments confirmed that the proposed technology can improve the accuracy of query cost estimation as aging is incurred in the database.

As a first step, this paper has focused on a careful but fundamental investigation of our purposed approach. Many open problems still remain. First, we would like to extend our technical investigation toward database queries of higher complexity. Second, we would like to extend our experiments by using different real-world data sets and queries in order to validate the benefits for a wide spectrum of applications. Finally, we also plan to work on implementation of the proposed framework into PostgreSQL so as to share our knowledge among the community.

References

1. Heyman, D.P.: Mathematical models of database degradation. ACM Trans. Database Syst. (TODS) **7**(4), 615–631 (1982)
2. Sockut, G.H., Goldberg, R.P.: Database reorganization-principles and practice. ACM Comput. Surv. (CSUR) **11**(4), 371–395 (1979)
3. Shneiderman, B.: Optimum data base reorganization points. Commun. ACM **16**(6), 362–365 (1973)
4. Bing Yao, S., Sundar Das, K., Teorey, T.J.: A dynamic database reorganization algorithm. ACM Trans. Database Syst. (TODS) **1**(2), 159–174 (1976)
5. Sockut, G.H., Beavin, T.A., Chang, C.-C.: A method for on-line reorganization of a database. IBM Syst. J. **36**(3), 411–436 (1997)
6. Omiecinski, E., Scheuermann, P.: A global approach to record clustering and file reorganization. In: Proceedings of the Seventh Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 201–219. British Computer Society (1984)
7. Kitsuregawa, M., Goda, K., Hoshino, T.: Storage fusion. In: Proceedings of the 2nd International Conference on Ubiquitous Information Management and Communication (ICUIMC2008), pp. 270–277. ACM (2008)
8. Ghandeharizadeh, S., Gao, S., Gahagan, C., Krauss, R.: An on-line reorganization framework for SAN file systems. In: Manolopoulos, Y., Pokorný, J., Sellis, T.K. (eds.) ADBIS 2006. LNCS, vol. 4152, pp. 399–414. Springer, Heidelberg (2006)
9. Chaudhuri, S.: An overview of query optimization in relational systems. In: Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 34–43. ACM (1998)
10. Jarke, M., Koch, J.: Query optimization in database systems. ACM Comput. Surv. (CsUR) **16**(2), 111–152 (1984)
11. Graefe, G.: The cascades framework for query optimization. Data Eng. Bull. **18**(3), 19–29 (1995)
12. Waas, F.M., Hellerstein, J.M.: Parallelizing extensible query optimizers. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 871–878. ACM (2009)
13. Chen, C.M., Roussopoulos, N.: The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. Springer, Heidelberg (1994)
14. Perez, L.L., Jermaine, C.M.: History-aware query optimization with materialized intermediate views. In: 2014 IEEE 30th International Conference on Data Engineering (ICDE), pp. 520–531. IEEE (2014)
15. Haas, L.M., Carey, M.J., Livny, M., Shukla, A.: Seeking the truth about ad hoc join costs. The VLDB J. **6**(3), 241–256 (1997)
16. Ghodsnia, P., Bowman, I.T., Nica, A.: Parallel I/O aware query optimization. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 349–360. ACM (2014)