HOLISTIC OPTIMIZATION BY PREFETCHING QUERY RESULTS

Bodepudi Sai Purna Chand[1], Y Vijayalata[2], SK Althaf Hussain Basha[3]
[1]PG Scholar, Dept. of IT, Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad.
[2]Professor and Head, Dept. of IT, Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad.
[3]Professor and Head, Dept. of IT, Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad.

***Abstract:** This paper perform the optimizing performance of database applications by means of automatically prefetching query results. The importance of this paper is for time estimation of executed cost and analyzing execution plan and optimizing queries. Incorporated techniques performed into a tool for SQL Server, to prefetch query results in Java programs that use JDBC. In this we perform the prefetching opportunities and chaining with algorithms. In many cases, the query is in a procedure which does not offer much scope for prefetching within the procedure; in contrast, our approach can perform prefetching in a calling procedure, even when the actual query is in a called procedure, thereby greatly improving the benefits due to prefetching.*

***Keywords:** Performance, Prefetching, Optimize, Cost, Time*

1. Introduction

In many cases, the query is in a procedure which does not offer much scope for prefetching within the procedure; in contrast, our approach can perform prefetching in a calling procedure, even when the actual query is in a called procedure, thereby greatly improving the benefits due to prefetching. The Paper mainly focus on the Execution time Graph at lower cost of each node. The algorithms of prefetching also used for estimation time. Performing these actions synchronously results in a lot of latency since the calling application blocks during stages .Much of the effects of latency can be reduced if these mappings are overlapped with local computations or other requests. Such overlap mappings can be achieved by issuing asynchronous requests in advance, while the application continues performing others task. In many use cases, the results can be made available by the time they are actually required, thereby completely hiding the effect of latency. This idea of making query results available before they are actually needed by the application, is called query result prefetching. In this paper we use softwares java with NetBeans and Derby as Driver of JDBC and in Sql server with Optimizing SQL Server Query Performance at a glance:

- Analyzing execution plans
- Optimizing queries
- Identifying queries to tune Our technical contributions in this paper are as follows:

1. We give a algorithm which statically inserts prefetch instructions at the earliest possible point across procedure calls, in presence of conditional branching and loops.
2. We propose enhancements such as code motion, chaining, and rewriting prefetch requests to increase benefits of prefetching.

2. Related Work

The idea of prefetching has been used in many areas of computers. Prefetching has long been supported for device IO in many operating systems, especially when it is sequential IO as in [11]. Databases internally use prefetching extensively to improve performance of query processing like MYSQL, SQL SERVER as in [9]. Even if the access pattern is not strictly sequential, it exhibits spatial locality in many cases, and prefetching is achieved by fetching databases or pages at a time. There has been earlier work where the prefetch is not based on physical layout and spatial locality, but on request patterns. More recently, approaches based on static analysis have been proposed to address problems with similar goals and calls. For every query, they place a copy of all variable initializations that the query uses directly or indirectly (through some other variable) at the beginning of the program. Next, they put a non-blocking execute function call for the query as in [2] after all these variable initializations. However, as we demonstrate in this paper, this problem requires a detailed analysis of the program. Firstly, placing copies of all variable initializations at the beginning of the program may not only duplicate many computations, but worse, it can lead to incorrect behaviour in the presence of side effects, global variables, local variables and conditional assignments. Secondly, they do not consider inter procedural prefetch, which restricts the benefits of their algorithm. There has been earlier work where the prefetch is not based on physical layout and spatial locality, but on request patterns. The idea of prefetching has been used in many areas and supported for device in operating systems, Sequential scans can be speeded up to a large extent by prefetching even if the access pattern is not strictly sequential, it exhibits spatial locality in many cases and prefetching is achieved. Next, they put a non-blocking execute function call for the query after all these variable initializations. However, as we demonstrate in this paper, this problem requires a detailed analysis of the program. Also, as in [1] batching may not be applicable altogether when there is no set-oriented interface for the request invoked. Our work guarantees correctness and places prefetches at the earliest possible point across method calls. In our earlier work, we proposed program transformation methods to exploit set oriented query execution or asynchronous submission to improve performance of iterative execution of parameterized queries as in [12]. Although batching reduces roundtrip delays and allows setoriented execution of queries as in [5], it does not overlap client computation with that of the server, as the client completely blocks after submitting the batch. Also, batching may not be applicable altogether when there is no set-oriented interface for the request invoked the techniques proposed here do not depend on loop fission, although as discussed in the two approaches can be used together for maximum benefit.

3. Proposed Work:

The Analyzing Execution Plans is when tuning an individual query, usually start by looking at the execution plan of that query. The execution plan describes the sequence of operations, physical and logical, that SQL Server TM will perform in order to fulfill the query and produce the desired result set. The Optimizing queries is the first thing to understand about execution plans is how it is generated. SQL Server uses a cost-based query optimizer—that is, it tries to generate an execution plan with the lowest estimated cost. The estimate is based on the data distribution statistics that are available to the optimizer when it evaluates each table involved in the query. If those statistics are missing or outdated, the query optimizer will lack vital information that it needs for the query optimization process and therefore its estimates will likely be off the mark.The Identifying queries to tune now look at a simple query, its execution plan, and some ways to improve its performance. To execute this query using Management Studio with the Include Actual Execution Plan option turned on in the Adventure Works sample database on SQL Server 2005: Fig1, Fig2, Fig3 tells you the execution time cost and estimation.

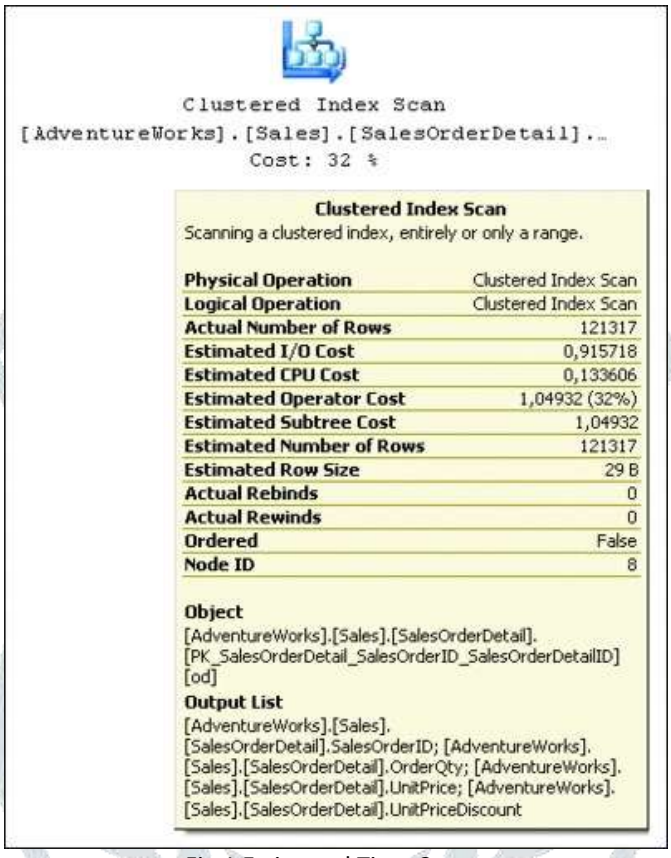


Fig.1 Estimated Time Cost

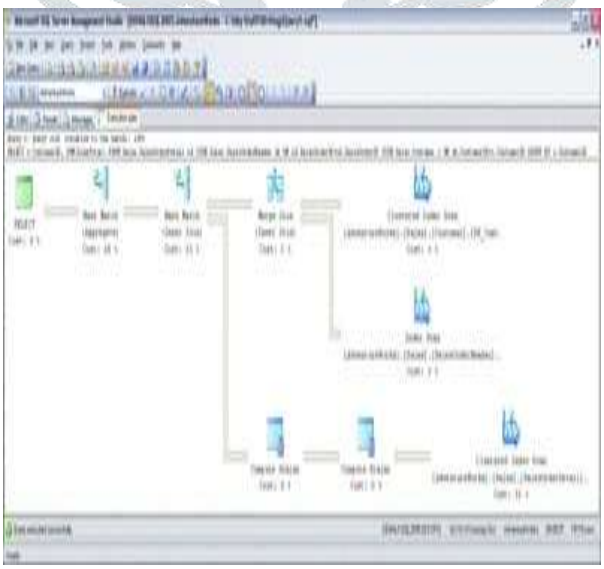


Fig. 2 Estimated Time and cost

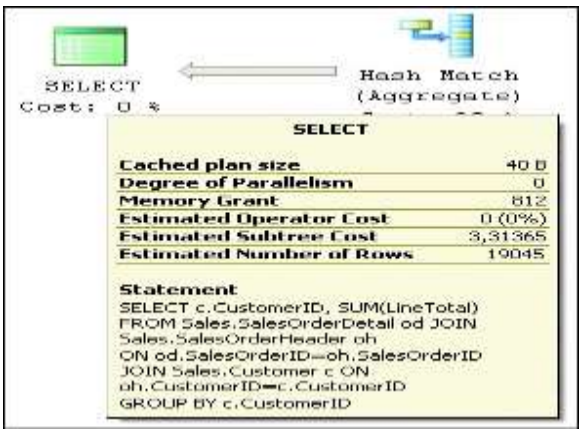


Fig.3 Total estimated execution cost of the query

Fig. 4 The execution plan after eliminating Customer table from the query

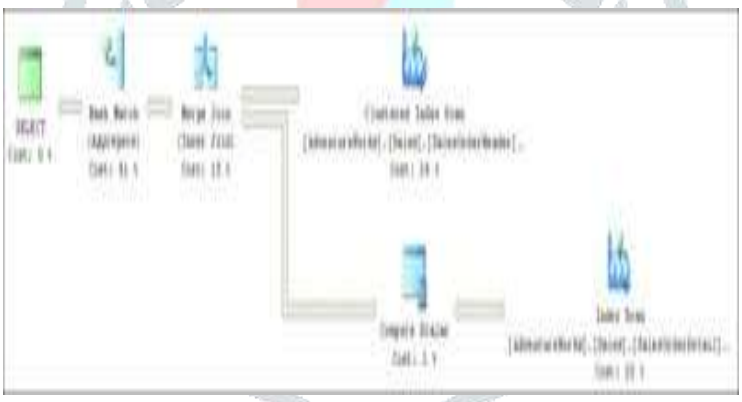


Fig.5 Optimized execution plan

Indexed Views:

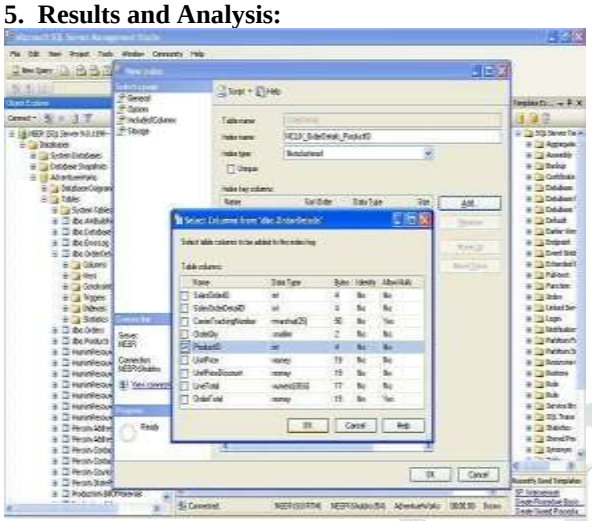
If the performance of my example query is very important, I can go one step further and create an indexed view that physically stores the materialized results of the query. Note the WITH SCHEMA BINDING option, which is a prerequisite for creating an index on such a view, and the COUNT_BIG(*) function, which is necessary if our index definition contains an aggregate function (in this example, SUM). After I create this view, I can create an index on it, like so:

Fig. 6 Execution plan when using indexed view

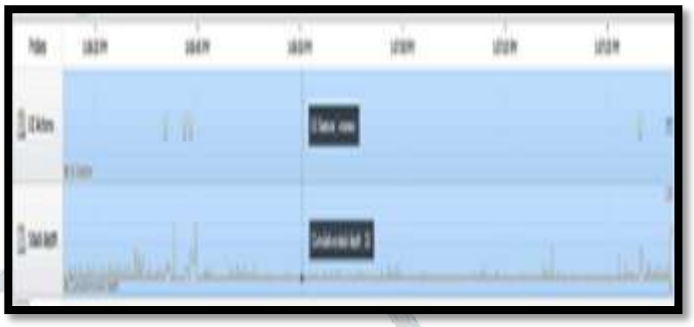
There are 3 steps for viewing Index in Databases are :

Step 1: Apply proper indexing in the table columns in the database Well, some could argue whether implementing proper indexing should be the first step in the performance optimization process for a database. But I would prefer applying indexing properly in the database.

5. Results and Analysis:



Graph Estimations are:



6. Conclusion and Future work:

Program analysis based approach to automatically detect Step 2: Create the appropriate covering indexes: opportunities for prefetching query results in database applications. So you have created all the appropriate indexes in your database, The algorithms presented in this paper significantly improve right? Suppose, in this process, you have created an index on a performance by prefetching across procedure calls, while avoiding foreign key column (ProductID) in wasteful prefetches. The future work is we propose a program the Sales(SelesID,SalesDate,SalesPersonID,ProductID,Qty) table. analysis based approach to automatically detect opportunities for Now, assuming that the ProductID column is a "highly selective" prefetching query results in database applications. We propose a column (selects less than 5% of the total number of rows rows using program analysis based approach to automatically detect any ProductID value in the search criteria), any SELECT query that opportunities for prefetching query results in database applications. reads data from this table using the indexed column (ProductID) in The algorithms presented in this paper significantly improve the where clause should run fast, right? performance by prefetching across procedure calls, while avoiding Step 3: Defragment indexes if fragmentation occurs: wasteful prefetches. We also propose enhancements that transform OK, you created all the appropriate indexes in your tables. Or, may programs and queries to increase applicability as well as benefits of be, indexes are already there in your database tables. But you might prefetching. Although we present our techniques in the context of not still get the desired good performance according to your database queries, they are more general in applicability. We present expectations. There is a strong chance that index fragmentation has a detailed experimental study, conducted on real world and occurred. benchmark applications, that show performance gains of more than

50% in many usecases. As part of future work, firstly, we plan to The algorithms of prefetch requests and Interprocedural prefetch complete our implementation to handle all the API methods of
also used for estimation time.They are briefly described as:

Prefetch requests:
1.remove all critical edges by edge splitting
2.perform Query AnticipabilityAnalysis on g w.r.t Q
3.append Prefetch Request(n, q)
4.prepend Prefetch Request(n, q)

Inter procedural Prefetch Requests:

1.Vertices of CG sorted in reverse topological order
2.run the modified intraprocedural algorithm
3.begin
4.remove(s,v) //remove s from procedure v
5.callSites = {cfg(src(e)) | e 2 CG and dest(e) == v}
6. replace formal parameters in s with their
7.actual counterparts in c
8.replaceParameters(s, c)
9.prependPrefetchRequest(s, t)

Hibernate, and to provide extensibility features enabling easy addition of any Web service API. We also plan to implement a more sophisticated cache manager, supporting standard replacement policies as well as invalidation of cached results. We also plan to make the decision of which calls to prefetch, and the program point where it needs to be placed in order to maximize benefit, in a cost based manner. The prefetching algorithm currently moves the prefetch instruction to call sites only if it can be pushed to the entry of a method. However, in many cases, there could be assignment statements that only the query depends on, which could also be moved to call sites along with the prefetch. This requires our code motion algorithm to be extended for the interprocedural case.

References:

[1] R. Guravannavar and S. Sudarshan, "Rewriting procedures for batched bindings," in Proc. Int. Conf. Very Large Databases, 2008 pp. 1107– 1123.

[2] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan, "DBridge: A program rewrite tool for set-oriented query execution," in Proc. IEEE 27th Int. Conf. Data Eng., 2011, pp. 1284–1287.

[3] K. Ramachandra, R. Guravannavar, and S. Sudarshan, "Program analysis and transformation for holistic optimization of database applications," in Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal., 2012, pp. 39–44.

[4] M. Chavan, R. Guravannavar, K. Ramachandra, and S.

[5] Sudarshan, "Program transformations for asynchronous query submission," in Proc. IEEE 27th Int. Conf. Data Eng., 2011, pp. 375–386.

[6] R. Guravannavar, "Optimization and evaluation of nested queries and procedures," Ph.D. dissertation, Dept. Comput. Sci. Eng., Indian Inst. Technol., Bombay, India, 2009. [6] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," ACM Trans. Program. Lang. Syst., vol. 9, no. 3, pp. 319–349, 1987

[7] K. Kennedy and K. S. McKinley, "Loop distribution with arbitrary control flow," in Proc. ACM/IEEE Conf. Supercomput., 1990, pp. 407–416.

[8] A. Manjhi, C. Garrod, B. M. Maggs, T. C. Mowry, and A.

[9] Tomasic, "Holistic query transformations for dynamic web applications," in Proc. IEEE 25th Int. Conf. Data Eng., 2009, pp. 1175–1178. [9] G. Graefe, "Executing nested queries," in Proc. 10th Conf. Database Syst. Business, Technol. Web, 2003.

[10] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi, "Execution strategies for SQL subqueries," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2007, pp. 993–1004.

[11] S. Iyengar, S. Sudarshan, S. Kumar, and R. Agrawal,

[12] "Exploiting asynchronous IO using the asynchronous iterator model," in Proc. Int. Conf. Manage. Data, 2008, pp. 127–138. [12] A. Dasgupta, V. Narasayya, and M. Syamala, "A static analysis framework for database applications," in Proc. IEEE Int. Conf. Data

[13] Eng., 2009, pp. 1403–1414. K. Ramachandra and S. Sudarshan, "Holistic optimization by prefetching query results," in Proc. ACM SIGMOD Int. Conf. Manage. Data, 2012, pp. 133–144