Automatic Workload Driven Index Defragmentation

Vivek Narasayya
Microsoft Research
Redmond, WA, USA
viveknar@microsoft.com

Hyunjung Park†
Stanford University
Stanford, CA, USA
hyunjung@cs.stanford.edu

Manoj Syamala
Microsoft Research
Redmond, WA, USA
manojsy@microsoft.com

ABSTRACT

Queries that scan a B-Tree index can suffer significant I/O performance degradation due to index fragmentation. The task of determining if an index should be defragmented is challenging for database administrators (DBAs) since today's database engines offer no support for quantifying the impact of defragmenting an index on query I/O performance. Furthermore, DBMSs only support defragmentation at the granularity of an entire B-Tree, which can be very restrictive since defragmentation is an expensive operation and workloads typically access different ranges of an index non-uniformly. We have developed techniques to address the above two challenges, and implemented a prototype of automatic workload driven index defragmentation functionality in Microsoft SQL Server. We demonstrate this functionality by showing (a) how the system tracks the potential benefit of defragmenting an index on I/O performance at low overhead, (b) the ability to defragment a range of a B-Tree index online, and (c) how the cost/benefit trade-off can be controlled in a policy driven manner to enable automatic workload driven index defragmentation requiring minimal DBA intervention.

1. INTRODUCTION

Decision support queries involve scanning large amounts of data. This data is typically stored in indexes, and thus the I/O performance of such queries crucially depends on *fragmentation* in the index. Typically when an index is created there is little or no fragmentation, and the I/O performance of queries that scan the index is good. However, as data is inserted, updated and deleted, an index can get fragmented over time. There are two kinds of fragmentation, both of which can have significant impact on I/O performance of a query (see Figure 1 for example). *Internal* fragmentation occurs when leaf pages of an index are only partially filled, thus increasing the number of pages that need to be scanned. For example, page 101 in Figure 1a is only partially filled, containing two empty slots. *External* fragmentation occurs when the logical order of leaf pages in the B-Tree differs from the physical order in which the pages occur in the data file, thereby increasing the number of disk seeks required. A previous study [1] has shown that fragmentation can reduce the I/O performance of decision support queries significantly (e.g. by 5x).

† Work done while author was visiting Microsoft Research

Proceedings of the VLDB Endowment, Vol. 4, No. 12

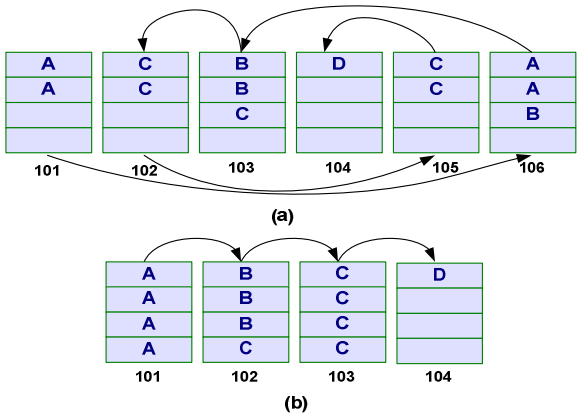

Figure 1. (a) Example of internal and external fragmentation in a B-Tree index. (b) Same index after it is defragmented.

Today's DBMSs offers techniques to gather fragmentation information of an index. For example, Microsoft SQL Server has virtual tables that report external and internal fragmentation statistics for an index. While such statistics can quantify fragmentation levels, there is no support in today's engines to quantify the *benefit* (i.e. reductions in I/O for a query) if an index were to be defragmented. Furthermore, defragmenting an index is an expensive operation. For example, an index that is heavily fragmented may have very few queries in the workload that scan it. Defragmenting such an index would bring very little benefit in terms of reducing I/Os but can incur the high cost of defragmentation.

Another significant limitation of today's DBMSs is that they only support defragmentation at the *granularity* of an entire B-Tree. This makes index defragmentation an expensive operation (e.g. they can incur significant I/O cost due to data movement and logging). Queries in the workload often access different ranges of an index non-uniformly. Thus, in principle, most of the performance benefit from defragmenting the index could be obtained by defragmenting only a small portion (i.e. range) of the index, but at a much lower cost. Furthermore, the ability to defragment smaller ranges allows defragmentation to be done incrementally and in an online manner, which is useful when the maintenance windows are small or non-existent.

For the reasons described above, determining if an index should be defragmented or not can be a difficult task for database administrators (DBAs). Furthermore, database-as-a-service (DaaS) offerings (e.g. SQL Azure [2]) have recently emerged. In such environments the service provider is typically responsible for performing index maintenance tasks such as defragmentation. Thus in both a traditional database as well as in a DaaS, the ability for the engine to automatically detect and correct performance problems due to index fragmentation can be valuable.

2. OVERVIEW OF SOLUTION

We have developed techniques to address the problems described above, and implemented a prototype of our solution in Microsoft SQL Server. Below we briefly describe the functionality of each component in our solution.

RangeTracker: We have developed a new monitoring component in the database engine that can estimate the benefit of defragmenting an index (or a range of the index) for the queries that have executed on the system. In particular, it estimates the reduction in the number of I/Os for a query that scans an index that would result if that index were to be defragmented. This "what-if" analysis is a key to making an informed decision on whether an index should be defragmented. We show how such monitoring can be done at low overhead by piggybacking on execution of queries in the system.

RangeDefragmenter: We have developed a new mechanism for defragmenting a range of an index. This mechanism can be invoked online, i.e., it can be invoked with minimal locking, thereby allowing concurrent queries and updates to proceed without significant blocking. A key advantage of such range level defragmentation is that most of the benefits of defragmentation for a query (or workload) can often be realized by only defragmenting a small part of the entire index.

Defragmentation Policy: We have developed a policy for automatically deciding when an index (or range of an index) should be defragmented. This policy takes into account the *benefit* of defragmentation as well as the *cost*. Intuitively, the policy looks for "sufficient evidence" based on the workload before triggering defragmentation of an index. The policy can be configured by a DBA in different ways, e.g., how aggressive or conservative the system should be, at what time of day defragmentation can be invoked, etc.

More technical details of our work can be found in [3]. These techniques are novel and to the best of our knowledge this functionality does not exist in any commercial DBMS.

3. DEMONSTRATION

The goal of the demonstration is to highlight the importance of the index defragmentation problem, and the effectiveness of our techniques in addressing the index defragmentation problem. In particular, we show the functionality of each of the components described above: *RangeTracker*, *RangeDefragmenter* as well as the *Defragmentation Policy*. Our demo will be based on a prototype we developed in Microsoft SQL Server 2008 R2 edition. The demo also involves a client driver program that (a) executes queries and updates against our server, (b) visualizes the I/O performance of queries, and (c) visualizes the relevant metrics used internally by the server in our solution (e.g. benefit of defragmenting a range, cost of defragmenting a range).

We now outline the demo scenario using a sequence of screenshots. Figure 2 shows a screenshot of the client driver program. The client driver allows us to choose a workload and also specify other input parameters like the number of times the query needs to execute. The output grid at the top shows the workload activity (Select, Insert, Delete, Update) in timestamp order and fragmentation metrics per index. It also shows the number of reads/writes issued by the workload. The chart on the left represent the actual number of I/O's per activity (i.e., query). The chart on the right shows the estimated *cost* of defrgamenting

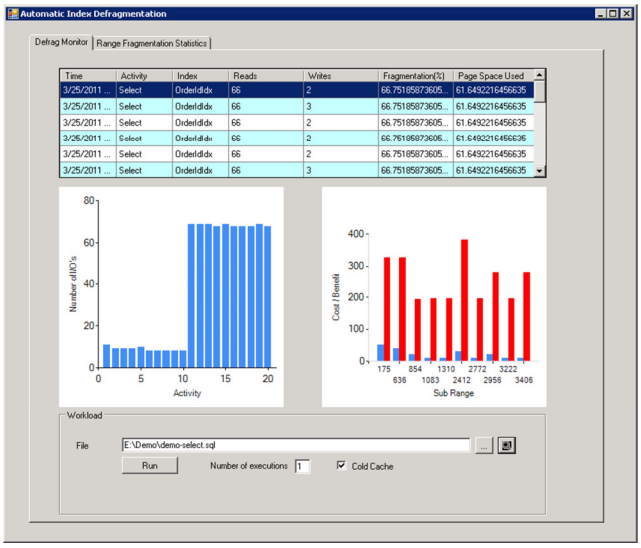

Figure 2. Screenshot of client driver program showing (a) I/O performance of each query, (b) benefit and cost of defragmenting each range of the index, and (c) queries, updates that have executed on the server.

a range of the index (red bar) and the estimated *benefit* of defragmenting the range (blue bar).

Figure 2 shows the state of the server after the following sequence of activity: (i) 10 range scans on an unfragmented index, (ii) Insert and Delete transactions that fragment the index, (iii) 10 range scans (same query). Observe that the number of I/Os required for the same query is now much higher (by about 7x) due to the increased data size and due to the fact that the index is much more fragmented. Also note that in the second chart, we begin to see the benefit and cost of defragmentation beginning to accumulate for ranges of the index. These benefits and costs are comptued by the *RangeTracker* module. At this point the costs are significantly higher than the benefits.

The next screenshot, shown in Figure 3, shows the state of the server after the same query has been executed several more times. We now notice that for a few index ranges the benefits have accumulated significantly. The automated *Defragmentation Policy* to decide if an index range should be defragmented is configurable. For simplicity, in the demo, we configure a range to be defragmented if its benefit exceeds the cost. Observe that in Figure 3, for a couple of ranges, the benefit is almost equal to the cost.

Figure 4 shows a screenshot of the situation after the same query has further been executed a few more times. At this point, a couple of ranges have already been automatically defragmented by the system (as controlled by the policy above). We also note that the I/O cost of queries has begun to reduce due to the fact that the ranges have been defragmented.

Figure 5 shows the state of the server after the scan query was further run a few more times. At this point most of the index ranges have been automatically defragmented, and the I/O cost of queries that scan the index has reduced significantly. Note that final I/O cost of the query is higher than the initial cost. This is because the initial Insert operations (that caused the fragmentation) added a significant amount of new data – thus there is more data to be scanned).

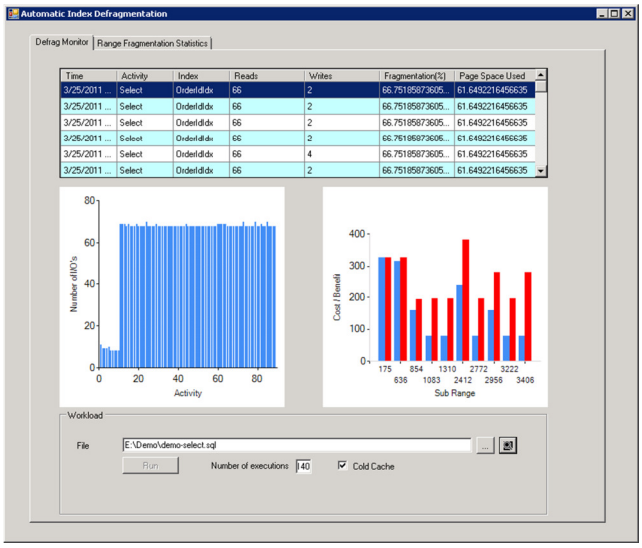

Figure 3. Screenshot showing how benefit accumulate for each index range as more queries execute on the server.

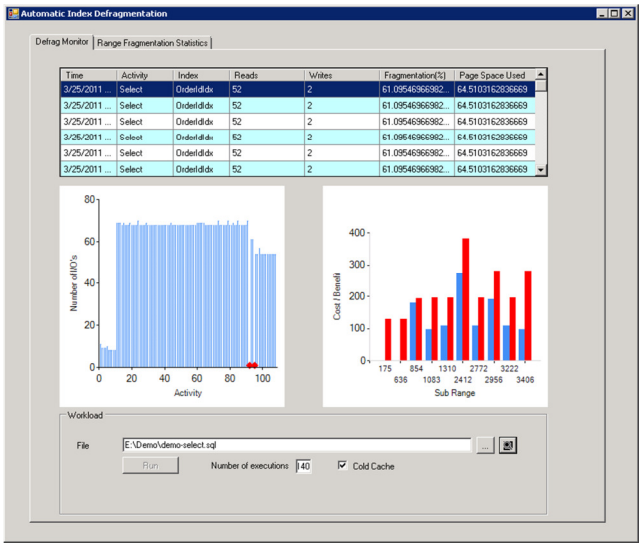

Figure 4. Screenshot showing how the I/O cost of queries begins to reduce after a couple of index ranges were automatically defragmented. The red dots on the left chart denote that an index range was defragmented.

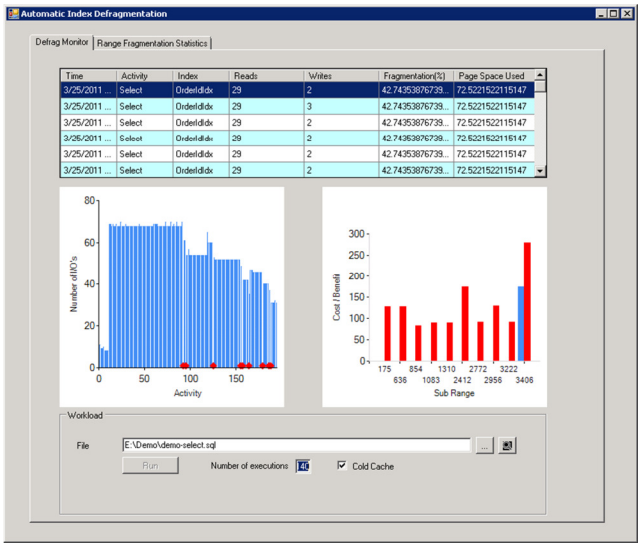

Figure 5. Screenshot showing query I/O performance after multiple index ranges have been defragmented.

4. CONCLUSION

Index defragmentation is an important problem in database systems, and is becoming even more important to address automatically in emerging cloud data services where it is not possible to have expert database administrators for each database. We have developed techniques for automatically detecting when it is beneficial to defragment an index and a mechanism for actually defragmenting a range of an index in an online manner. We show how to put these mechanisms together in a policy driven manner that requires minimal DBA intervention. This functionality is novel, and we have implemented our solutions in a real world commercial DBMS: Microsoft SQL Server. This is the first demonstration of this technology outside of Microsoft.

5. REFERENCES

[1] M. Ruthruff. Microsoft SQL Server 2000 Index Defragmentation Best Practices. Microsoft TechNet, 2003. http://technet.microsoft.com/en-us/library/cc966523.aspx

[2] Microsoft SQL Azure. http://www.microsoft.com/sqlazure

[3] V. Narasayya and M. Syamala. Workload Driven Index Defragmentation. In *ICDE*, pp. 497-508, 2010.