Workload Driven Index Defragmentation

Vivek Narasayya, Manoj Syamala

Microsoft Research
One Microsoft Way, Redmond, WA, USA.
viveknar@microsoft.com
manojsy@microsoft.com

Abstract—**Decision support queries that scan large indexes can suffer significant degradation in I/O performance due to index fragmentation. DBAs rely on rules of thumb that use index size and fragmentation information to accomplish the task of deciding which indexes to defragment. However, there are two fundamental limitations that make this task challenging. First, database engines offer little support to help estimate the impact of defragmenting an index on the I/O performance of a query. Second, defragmentation is supported only at the granularity of an entire B+-Tree, which can be too restrictive since defragmentation is an expensive operation. This paper describes techniques for addressing the above limitations. We also study the problem of selecting the appropriate indexes to defragment for a given workload. We have implemented our techniques in Microsoft SQL Server and developed a tool that can provide appropriate index defragmentation recommendations to DBAs. We evaluate the effectiveness of the proposed techniques on several real and synthetic databases.**

I. INTRODUCTION

Decision support queries often require scans of large indexes. When data is inserted or updated, indexes on that table can get fragmented due to page splits in the B+-Tree. There are two kinds of index fragmentation, both of which can have significant impact on I/O performance of a query. *Internal* fragmentation occurs when a leaf page of an index is only partially filled, thus increasing the number of pages that need to be scanned. *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. Thus, compared to an index that is not fragmented, both internal and external fragmentation can result in more I/Os for queries that scan the index. For example, the article [12] shows that fragmentation can reduce the I/O performance of decision support queries significantly (e.g. by a factor of 5 in the execution time).

Today's relational database management systems (DBMSs) support mechanisms for an index to be *defragmented*. Defragmenting an index consists of compacting pages to reduce internal fragmentation, as well as reordering pages to reduce external fragmentation. Index defragmentation is a heavyweight operation that can itself incur significant I/O cost, and must therefore be invoked judiciously. The responsibility of deciding which indexes in the database to defragment typically falls on a database administrator (DBA). To assist DBAs in this task, DBMSs expose mechanisms for reporting information about the current fragmentation of each index, including measures of internal and external fragmentation. We refer to techniques that rely only on the fragmentation information for deciding which indexes to defragment as *data-driven* approaches. Since it is usually not possible to defragment all indexes within a typical batch window (e.g. a few hours at night), DBAs today use rules of thumb to select which indexes to defragment, e.g. select indexes that are most fragmented.

We observe two key limitations with the state-of-the-art in index defragmentation. First, the granularity at which index defragmentation is supported is the *full* B+-Tree, which can very expensive for large indexes. In many cases, the fragmentation may not be uniformly distributed across the B+Tree. For example, consider a clustered index on the *OrderDate* column of a large fact table that stores order information. As new data is inserted into the fact table, the B+-Tree gets fragmented. However, it is often the case that many of the queries referencing only recent data (e.g. last month or last quarter). The performance benefit from defragmenting the index only arises for ranges scanned by queries. Thus the ability to perform index defragmentation for a specified logical range on the key column of a B+-Tree (e.g. *OrderDate* > '06/30/2009') can potentially provide most of the benefits of defragmenting the full index but at a much lower cost. We refer to this capability as *range-level index defragmentation*.

A second limitation is that while data-driven approaches to index defragmentation are easy to understand and implement, a purely data-driven approach can suggest defragmenting indexes that have little or no impact on query performance. This is because they ignore potentially valuable *workload* information, i.e. information about queries that scan the index. Using workload information can be crucial in large data warehouses consisting of hundreds of indexes, which is typical in enterprise applications. While leveraging workload information can be important, there are a couple of key challenges which make it difficult for DBAs to exploit workload information for index defragmentation. First, it is difficult to estimate the impact of defragmenting an index on the I/O performance of a query that scans that index. Naturally, it is important to estimate the impact *without* actually defragmenting the index. Such "what-if" analysis of impact of defragmentation on query I/O performance is an essential component for enabling a workload driven approach to index defragmentation. Second, even if the above "what-if" analysis functionality is available, selecting *which indexes to*

defragment for large databases (with many indexes) and workload (with many queries) can be non-trivial.

Despite the importance of index defragmentation, to the best of our knowledge, there is no prior published work that addresses the above challenges. Effective solutions to these problems can greatly reduce the DBA's burden and hence the cost of administering a DBMS. This paper makes the following contributions. First, we introduce the novel idea of range-level index defragmentation and describe the desirable properties that any range-level index defragmentation scheme should satisfy. We present a scheme that meets these requirements (Section IV). Second, we describe an API in the DBMS engine for efficiently supporting "what-if" analysis of the impact of defragmenting an index (or a range of the index) on a given query (Section V). Third, we formally define the problem of selecting an optimal set of indexes (or index ranges) to defragment, and show that this problem is computationally hard. We present an algorithm that, given a database, a workload of SQL queries, and a budget for the cost of defragmenting indexes, can automatically recommend which indexes (or index ranges) should be defragmented (Section VI). Finally, we have implemented the functionality described above in a commercial database system: Microsoft SQL Server 2008. We present results of experiments on both real and synthetic databases that highlight: (a) The importance of exploiting workload information for index defragmentation; (b) The advantages of range-level index defragmentation compared to full index defragmentation; (c) The effectiveness of our techniques for automatically recommending which indexes to defragment.

We begin by first reviewing (in Section II) the kinds of index fragmentation that can occur in a B+-Tree, the impact of fragmentation on I/O performance, and mechanisms for defragmenting an index. Section III formalizes the index defragmentation problem, outlines alternative approaches, and describes the architecture of our solution

II. PRELIMINARIES

We present the types of index fragmentation, their impact of I/O performance of a query, and mechanisms for index defragmentation.

A. Index Structures

All modern databases support index structures for speeding up access to data. An index I is defined by a sequence of columns on a given table or materialized view. In this paper, we assume that indexes are stored as B+-Trees. In general, when an index is partitioned, each partition of the index is stored as a separate B+-Tree. For ease of exposition, we assume that each index has only one partition; however the techniques in this paper carry over to the case of multiple partitions as well. Thus in this paper we will use the terms index and B+-Tree interchangeably.

Note that in a B+-Tree, the leaf pages form a linked list where each leaf page points to the next leaf page in the logical order of keys in the index. This makes range queries that scan the index more efficient. Figure 1 (a) shows the leaf pages of a B+Tree as a *singly* linked list. In general, the leaf pages are often connected via a doubly linked list which can improve the efficiency of both ascending and descending order range queries.

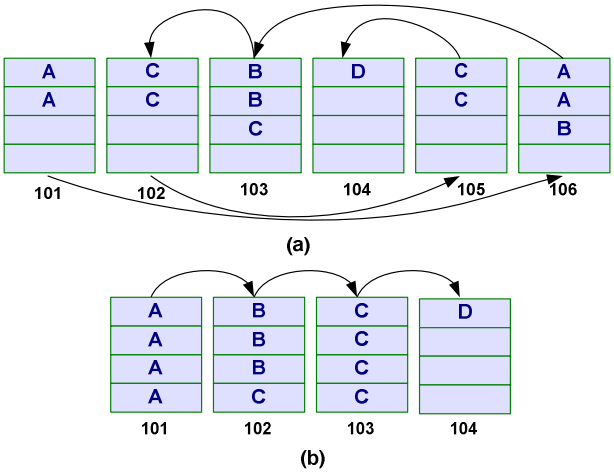

Figure 1. (a) Leaf nodes of a B+-Tree with internal and external fragmentation. (b) B+-Tree after it is defragmented.

B. Index Fragmentation

When an index is first built, there is little or no fragmentation. Over time, as data is inserted, deleted and updated, index fragmentation can increase due to B+-Tree page splits. There are two important types of fragmentation in an index.

1) Internal Fragmentation: Occurs when pages in the index are not filled to their maximum limit. The maximum limit can be specified by the DBA. The system usually sets a default (e.g. 0.85). Without loss of generality, in this paper we will assume a fill factor of 1.0 for simplicity of exposition. Our techniques and results carry over for any given value of fill factor. Consider the index shown in Figure 1(a). If we assume that each page holds 4 rows, then all pages in the index have internal fragmentation since each page has unused space page.

2) External Fragmentation: Occurs when leaf pages of the index are not in logical order. This can happen for instance when new data is inserted into a page that does not have sufficient space to hold the new row. This causes a B+-Tree split and a new page to be allocated which causes the pages to be out of order. Consider the index shown in Figure 1(a). The logical order of the pages in the B+-Tree, which can be obtained by following the B+-Tree pointers is: 101, 106, 103, 102, 105, 104. The physical order of pages in the B+-Tree is 101, 102, 103, 104, 105, 106. Since these two orders are not identical, the index has external fragmentation.

C. Impact of Index Fragmentation on Query Performance

Index fragmentation can significantly affect I/O performance of queries that scan the index. Internal fragmentation results in more pages than necessary to store the data. This implies more I/Os required to scan the index. When there is external fragmentation, the physical order of pages in the data file does not match the logical order of pages in the B+-Tree. In this case, more I/Os must be issued to seek

to different locations in the file. There are a few situations in which index fragmentation has little on no impact on query I/O performance. First, if the query is doing a lookup that requires accessing only a single leaf page of the index, then fragmentation has no impact. Such queries are common in OLTP workloads. Second, when (some of the) leaf pages of the index required by the query are already resident in the DBMS buffer pool, fewer I/Os on the index are required. Hence the impact of fragmentation on I/O performance of the query is reduced.

D. Index Defragmentation

There are two commonly used approaches available for defragmenting an index: (a) *Rebuild* (b) *Reorganize.*

Rebuilding an index involves the same process as creating an index. It requires a full scan and sort of the rows on which the index is defined. The DBMS then allocates and writes out the pages of the index. Each leaf page is filled (modulo the specified fill factor) and thus there is no internal fragmentation. Since the pages are written to the file sequentially in logical order, there is also no external fragmentation.

In contrast to rebuilding, reorganizing an index (we refer to this operation as *Reorg*) is an *in-place* operation and therefore does not require additional space. In the first phase (the *compaction* phase), internal fragmentation is removed by moving rows across pages. For example, in Figure 1(a), the two rows with value A from page 106 are moved to page 101 during the compaction step. Thus the number of pages in the index can reduce. In the second phase (the *swap* phase), the pages that remain after the compaction step are reordered via a series of page swaps so that the logical order of pages in the B+-Tree agrees with the physical order of pages in the data file.

Finally, although *Rebuild* and *Reorg* incur different costs, both are heavyweight operations that can involve a significant number of I/Os. In this paper, when we refer to defragmentation we will mean the *Reorg* operation. The techniques described in this paper, except Section IV, apply to both operations. In principle, the ideas of Section IV can also be extended for *Rebuild*, although we do not focus on that in this paper.

E. Definitions and Notation

Index Layout: We define the *layout* of an index to be the *sequence* of (*pid*, *c*) values of the leaf pages of the index in logical order; where *pid* is the page id of the leaf page, and *c* is the fraction of used space within the page. Thus assuming each page can hold 4 records, the layout for the index I in Figure 1(a) is: <(*101, 0.5),(106,0.75),(103,0.75), (102,0.5), (105,0.5), (104,0.25)*>.

N_I: Denotes number of pages in index I.

Reorg(I): Denotes the index that results by executing the reorganization operation for index I.

Compaction Ratio: The compaction ratio is a measure of the internal fragmentation of an index. It is the number of pages in the index if it is defragmented to the number of pages in the index currently. We denote the compaction ratio for an index I by *CR*(I). Note that $0 < CR(I) \leq 1$. $CR(I) = 1$ implies the index has no internal fragmentation, whereas a value close to 0 implies large internal fragmentation. Note that compaction ratio can be written using the above notation as:

$$CR(I) = \frac{N_{Reorg(I)}}{N_I}$$

External Fragmentation: *EF*(I) denotes a measure of external fragmentation of an index. *EF*(I) is the ratio of the number of fragments (non-contiguous of page sequences) in the index I to the total number of pages in the index.

ReorgCost(I): Denotes the cost of executing *Reorg*(I) as determined by a cost model. The main purpose of this cost model is to help differentiate (i.e. compare) defragmentation costs across different indexes. In this paper we use a cost model that captures the I/O cost of this operation. In particular we model the cost of defragmenting an index I using the *Reorg* operation using the following formula:

$$ReorgCost(I) = k_1 N_I + k_2 N_I\big(1 - CR(I)\big) + k_3 CR(I) N_I EF(I)$$

The first two terms in the above formula represents the cost of removing internal fragmentation. This involves scanning the leaf pages of the index to detect the amount of internal fragmentation, moving rows to fill pages, and removing unused pages (which is proportional to the amount of internal fragmentation). The final term represents the cost of removing external fragmentation. This step needs to work on $CR(I) \times N_I$ pages, since this is the number of pages remaining after internal fragmentation is removed. The cost of this step depends on the degree of external fragmentation *EF*(I). We have set the constants k_1, k_2, and k_3 by calibrating this cost model for our system (Microsoft SQL Server).

Index Ranges: As mentioned earlier, in this paper we will consider defragmenting *logical ranges* of an index. A logical range of an index is a range predicate on the leading column of the index. For an index with the leading column *OrderDate*, an example of a logical range is '06-30-2009' ≤ *OrderDate* ≤ '09-30-2009'. In general, it is possible to defragment a *set* of logical ranges R of an index. We will use *Reorg*(I, R) to denote that the set of ranges R of index I are defragmented. *Reorg*(I) is shorthand that denotes the special case where the full index is defragmented.

Workload: We define a workload **W** as a set of SQL query and update statements, where each statement has an associated weight. Thus $\mathbf{W} = \{(Q_1, w_1), \ldots (Q_n, w_n)\}$, where Q_i is the SQL statement, and w_i is the weight. For example, the weight can represent the frequency of a query. All DBMSs have mechanisms to collect the workload by monitoring the statements that execute against the database server (e.g. in Microsoft SQL Server, the Profiler provides this functionality).

III. INDEX DEFRAGMENTATION PROBLEM

A. Approaches to Index Defragmentation

As described in Section II defragmenting an index is an expensive operation. Furthermore, in many data warehouses, there are a large number of indexes which may be fragmented. DBAs typically have a limited time budget for defragmenting indexes (e.g. a nightly batch window). Thus, it is often not possible to defragment all indexes in the database.

In approaching the problem of deciding which indexes in the database to defragment, there are two orthogonal considerations: (a) Whether to leverage workload information or not. (b) Whether to allow range-level index defragmentation or not. Leveraging the workload can be important since we would like to pick indexes to defragment that have the most benefit on the I/O performance of the workload. Considering range-level index defragmentation is important since fragmentation can be skewed across different logical ranges of the index. Thus, defragmenting a logical range can potentially provide significant benefit for queries while incurring a small fraction of the cost of fully defragmenting the index. Thus, the four alternative approaches to index defragmentation are shown in Figure 2.

Full index defrag, Data only (**FULL**)	Range defrag, Data only (**RANGE**)
Full index defrag, Data + Workload (**FULL-W**)	Range defrag, Data + Workload (**RANGE-W**)

Figure 2. Approaches to index defragmentation.

FULL: This is a data-driven approach that only considers defragmenting the *full* index. In this approach, DBAs rely on APIs exposed by the DBMS that return measures of internal and external fragmentation for a B+-Tree (e.g. those mentioned in Section II.E). Typically, DBAs use rules of thumb/best practices to decide which indexes to defragment (e.g. indexes with the largest fragmentation).

FULL-W: This approach uses both data (i.e. fragmentation statistics of the index) as well as workload information. However, similar to FULL, it constrains that the entire index must be defragmented.

RANGE: This is similar to FULL in that it is a purely data-driven approach, but it leverages the ability to defragment a range of an index. Thus, it is strictly more general than FULL.

RANGE-W: The most general option that uses index fragmentation statistics as well as workload; and also leverages the ability to defragment a range. It uses strictly more information than both FULL-W and RANGE.

B. Problem Definition and Hardness

We now formally define the index defragmentation problem, and show that it is NP-Hard.

Index Defragmentation Problem: Given a database with a set of indexes $S = \{I_1, \ldots I_n\}$, a workload $W = \{(Q_1,w_1), \ldots (Q_m,w_m)\}$, where each Q_i is a query, and w_i is the weight of query Q_i. For each index I_j the cost of defragmenting the index is *ReorgCost*(Ij). The benefit of defragmenting index I_j on query Qi is *Benefit*(Q_i, I_j). Given a defragmentation cost budget of **B**, find a subset $D \subseteq S$ to defragment such that:

$$\sum_{d \in D} \sum_{i=1}^{m} w_i \times Benefit(Q_i, d)$$

is maximized, subject to the constraint:

$$\sum_{d \in D} ReorgCost(d) \leq \boldsymbol{B}$$

Claim: The Index Defragmentation Problem is NP-Hard.

Proof: See APPENDIX A. The reduction is from the Knapsack problem[11].

The above definition of the index defragmentation problem assumes the full index is defragmented. However, the formulation extends in a straightforward manner for the case when range-level index defragmentation is possible.

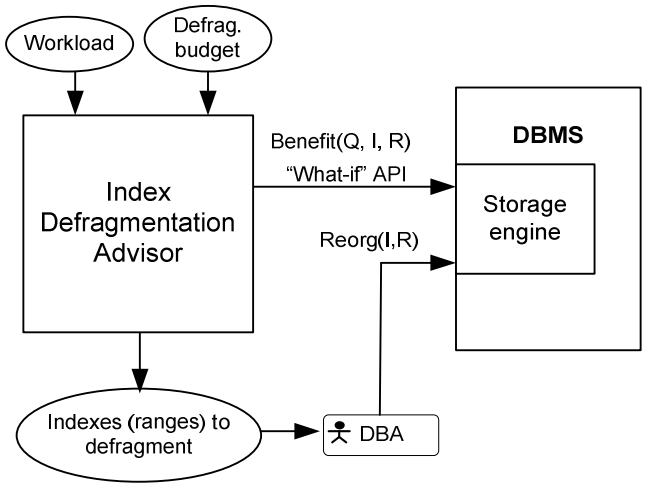

Figure 3. Architecture of the index defragmentation advisor tool.

C. Architecture Overview

The architecture of our solution to the index defragmentation problem is shown in Figure 3. We propose the following two key extensions to the storage engine of the DBMS: (a) Ability to perform range-level index defragmentation (Section IV). (b) A "what-if" API for estimating the impact of defragmenting an index (in general, a set of ranges of an index) on a given query (Section V). We also propose a client tool (an "Index Defragmentation Advisor") that can recommend a set of indexes (in general a set of index ranges) to defragmenting for a given workload and given budget on defragmentation cost.

IV. RANGE LEVEL INDEX DEFRAGMENTATION

As discussed in Section II, index defragmentation is a heavyweight operation that can involve a significant amount of I/O. However, DBMSs today only allow defragmentation at the level of the complete B+-Tree. Therefore, it is natural to consider whether it is possible to allow defragmentation at the level of a logical range of the B+-Tree rather than the

complete B+-Tree. For example, for the index shown in Figure 1(a), using range-level index defragmentation it would be possible to specify *Reorg*(I_r), which specifies that only the pages in the B+-Tree corresponding to the specified range *r* (e.g. *col* BETWEEN 'C' AND 'D', where *col* is the leading column of index I) should defragmented. After defragmenting this range, the index would appear as shown in Figure 4. Observe that other ranges such as *col* BETWEEN 'A' AND 'B' would not be defragmented.

There are two fundamental reasons why range level defragmentation can be advantageous. First, fragmentation may not be uniform across an entire index. It is common to have updates that are skewed towards certain key ranges of the indexes compared to other ranges. Thus, the fragmentation in the index can also be skewed. In such cases, defragmenting the range with large fragmentation may be adequate. Second, the workload may be skewed. For example, if most queries in the workload access a certain range of an index, then fragmenting that range may be sufficient.

In Section IV.A, we first outline the desirable properties that any range-level index defragmentation should possess. In Section IV.B we present a range-level index defragmentation method and show that it satisfies the desirable properties.

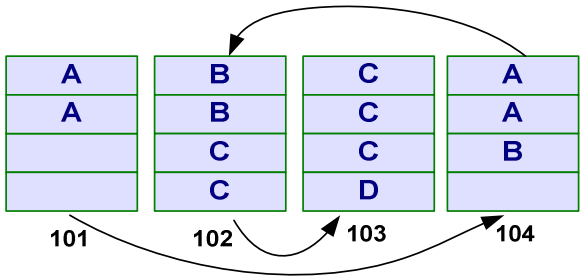

Figure 4. Index in Figure 1(a) after it has been defragmented for range: col BETWEEN 'C' and 'D'

A. Requirements for Range-Level Index Defragmentation

Consider a query that scans a range *r* of a given index I. We use *NumIOs*(*r*, I) to denote the number of I/Os required for the range scan *r* on index I. We present three key requirements that are desirable for any range-level defragmentation method.

Equivalence: The number of I/Os required for a query that scans exactly the defragmented range should be the same as though the full B+-Tree was defragmented. This requirement ensures that range-level defragmentation does not compromise I/O performance when compared to defragmenting the full index. Using the notation above, we can express the correctness requirement as:

$$NumIOs(r, Reorg(I, \{r\}) = NumIOs(r, Reorg(I))$$

Stability: Intuitively, this means that for any range that *we have already defragmented*, we do not want the defragmenting of a different range on the same index to "undo" the performance benefits. For example, suppose we defragment a range r_1, and suppose the number of I/Os required for a query that now scans r_1 is n_1. Now suppose we further defragment a range r_2. The number of I/Os required for a query that scans r_1 should still be n_1.

More formally, we can define stability as follows.

$$\forall\, r_1, r_2 \colon NumIOs(r_1, Reorg(I, \{r_1\})) = NumIOs(r_1, Reorg(I_1, \{r_2\})$$

where $I_1 = Reorg(I, \{r_1\})$.

Efficiency: The efficiency requirement states that the work done by the method to defragment a range should be proportional to the number of pages in the range. As an example, consider a partitioning of the index into two logical ranges. Ideally, the sum of the cost of invoking the range-defragmentation method on the two logical ranges should not cost significantly more than the cost of defragmenting the full index in a single invocation.

B. Range-Level Index Defragmentation Method

We now present a method for range-level index defragmentation that satisfies the above requirements. There are two key decisions that influence a range-level defragmentation method: (a) How the fragmentation of pages in the specified range *r* is removed. (b) The *offset*, measured in the number of pages, at which the first page of the range *r* (after defragmentation) will be placed. Note that the rest of the pages of *r* after defragmentation will be placed contiguously following the first page.

First, we observe that the *equivalence* and *efficiency* requirements described earlier determine how we proceed with respect to decision (a). In particular, our method will need to perform compaction and reordering of pages (see Section II.D) that belong to the specified range *r*. Thus the key remaining decision is (b), i.e. we need to determine the *offset* (in number of pages) from the first leaf page of the B+-Tree where the first page of the given range *r* should be placed after defragmentation. We use the *stability* requirement described earlier to guide our answer to this question.

To illustrate why this decision is non-trivial, consider the following example of a range level index defragmentation scheme that does *not* satisfy the stability requirement. It always set the *offset* to 0 regardless of the specified range *r*, i.e. place the pages of *r* starting at the beginning of the B+-Tree. It is easy to see that this scheme satisfies the equivalence and efficiency requirements. However, this scheme is not stable, since if we now defragment another logical range *r*' (e.g. it is non-overlapping with *r*), then the pages of *r*' get placed starting at offset 0. Thus, pages in *r* can get swapped to different locations in the file, and therefore *r* can become fragmented once again. In turn, this will increase the number of I/Os for any query that scans *r*, thereby violating stability.

1) Determining Offset

We now describe how we determine the offset. Let *p*(*r*) denote the *prefix* range of *r*, i.e. logical range between the first key in the index and the key just preceding the first key in range *r*. For example, in the index of Figure 4, if *r* is the range ['C','D'], then *p*(*r*) is the range ['A','B']. Similarly, let *s*(*r*) denote the suffix of range *r*. The following result tells us what offset should be used when defragmenting *r* in order to ensure stability.

Lemma: For any range level index defragmentation scheme $\mathfrak{I}$ that satisfies the equivalence and efficiency properties to be

stable, when defragmenting any range r on index I, it must use an offset equal to the number of pages in $p(r)$ in the fully defragmented index.

Proof: See APPENDIX **B**. (Using an adversary argument).

2) Efficiently Computing Offset

Based on the above result, our range-level defragmentation method needs to compute the offset for a range r as the number of pages in $p(r)$ (the prefix of r) *in the fully defragmented index*. Thus the key question is how to compute this value efficiently. Suppose the number of pages in $p(r)$ in the *current index* is N_c and the number of pages in $p(r)$ in the fully defragmented index is N_d. Note that $N_d \leq N_c$ and that the reduction (if any) occurs only due to internal fragmentation of the pages in $p(r)$. Thus:

$$N_d = N_c \times CR(p(r))$$

where $CR(p(r))$ is the compaction ratio of $p(r)$. This implies:

$$N_d = N_c \times \frac{\sum_{p \in p(r)} f_p}{N_c} = \sum_{p \in p(r)} f_p$$

where f_p is the fullness of page p in the range $p(r)$. Since computing f_p requires accessing the leaf page p it can be a potentially expensive operation. Thus the key challenge is to *accurately estimate* N_d without having to access each leaf page in $p(r)$.

Fortunately, the above estimation problem is amenable to the use of *sampling*. In particular, we compute f_p for a uniform random sample of the pages in $p(r)$ and scale up the results. The well-known theorem from [8] shows that the estimate thereby obtained is an *unbiased estimator* of the true value $\sum_{p \in p(r)} f_p$. Thus, if $\widehat{N_d}$ is an estimator of N_d, S is a uniform random sample of n pages in $p(r)$, and ε is the standard error of the estimator, then:

$$\widehat{N_d} = \frac{N_c}{n} \times \sum_{p \in S} f_p \qquad \varepsilon = \frac{N_c \times D \sqrt{1 - \frac{n}{N_c}}}{\sqrt{n}}$$

where D is the standard deviation of f_p values over all pages in $p(r)$. As we show in our experiments, using relatively small sampling fraction (such as 1%) is adequate to yield very accurate estimates of N_d. Thus, the offset computation can be made efficient.

RangeDefrag(r)
1. Estimate N_d using uniform random sampling over pages in $p(r)$, i.e., the prefix of r.
2. Traverse pages in r in logical order, and eliminate internal fragmentation by moving rows as needed. // compaction step
3. O = N_d pages from the start of the B+Tree
4. **For** each page p in r in logical order
5. If the page id of p is not already at offset O, swap p with the page currently at offset O
6. Advance O to the next physical page in the B+-Tree
7. **Return**

Figure 5. Algorithm for range-level index defragmentation

3) Algorithm

We summarize our overall algorithm for range-level index defragmentation in Figure 5. We first compute the offset at which the range will be placed. Step 2 is the compaction step, where the internal fragmentation for range r is eliminated, and Steps 3-6 eliminate external fragmentation. Observe that the main *overheads* of this method (relative to full index defragmentation) are incurred in Step 1. The other steps are necessary even in full index defragmentation. In Section VII.B we evaluate the efficiency and stability properties of our range-level index defragmentation method.

V. ESTIMATING IMPACT OF DEFRAGMENTATION ON I/O COST OF QUERY

A workload driven approach to index defragmentation requires the ability to quantify the impact of defragmenting the index on the I/O cost of a query. Since defragmenting an index is an expensive operation, it is important that the above functionality be supported *without* having to actually defragment the index and execute the query. If such a "what-if" analysis capability were available as an API in the DBMS, DBAs could benefit from it directly. For example, a DBA can compare two existing indexes by quantifying the impact of defragmenting each of them on queries in the workload. Furthermore, this API can also be leveraged by tools (such as the one we describe in Section VI) that provide automated recommendations for which indexes to defragment for a given workload of queries.

In this section, we describe how the above "what-if" analysis API can be implemented in a DBMS. We first describe this for the special case when the *full* index is defragmented. This special case is in fact important, since today's DBMSs only support full index defragmentation. We denote to the reduction in number of I/Os for a range scan query Q if index I is fully defragmented as ***Benefit*(Q, I).** Recall that in Section IV we introduced the idea of range-level index defragmentation. Thus, in the second part of this section, we discuss the necessary extensions to allow estimating the benefit for query Q if a *set of ranges* on I is defragmented (denoted by *Benefit*(Q, I, $\mathcal{R}$)).

A. Estimating Benefit when Full Index is Defragmented

The benefit of defragmenting an index I on a range scan query Q is the reduction in the number of I/Os for Q if the index is defragmented, i.e.

Benefit(Q, I) = *NumIOs*(Q,I) – *NumIOsPostDefrag*(Q, I)

where *NumIOs*(Q,I) is the number of I/Os required to execute the range scan Q over the index I, and *NumIOsPostDefrag*(Q,I) is the number of I/Os over the defragmented index I. We now describe how to compute each of the terms efficiently, i.e. without actually defragmenting the index or executing Q.

1) *Computing NumIOs*(Q, I)

The key observation that enables an efficient implementation of *NumIOs*(Q, I) is that to estimate the number of I/Os for a range scan query we only require the sequence of page ids in the B+-Tree scanned by the query. This information is available in the index pages of the B+-

Tree and does not require scanning the leaf pages. Since the ratio of index pages to leaf pages in a large B+-Tree index is typically very small, and the index pages of a B+-Tree are usually memory resident, the sequence of leaf page ids in a range can be obtained efficiently.

The pseudocode for our procedure *NumIOs* is shown in Figure 6. The idea is to "simulate" the execution of the range scan, but without issuing the actual I/Os. We maintain a lookahead buffer of page ids. The goal is to identify opportunities to identify contiguous page ids within this buffer, since such contiguous pages can be fetched using a single I/O, thereby optimizing access to disk. The size of the lookahead buffer (*M*) can be set to a value which is an upper bound of the number of pages that the system will fetch in a single I/O.

NumIOs(*Q, I*)
1. Let *r* be the range of the index I scanned by query Q.
2. Let **P** be a lookahead buffer that maintains leaf page ids in sorted order, max *M* elements.
3. n = 0;
4. **Do**
5. Add page ids to **P** obtained by iterating over index pages of B+-Tree in logical order until there are *M* page ids in **P** or last page id in range *r* is encountered.
6. Extract from **P** the maximal sequence of contiguous page ids at the start of **P**
7. n = n+ 1
8. **While**(**P** is non empty)
9. Return **n**.

Figure 6. Estimating the number of I/O for a query Q that scan a range of index I.

The algorithm first traverses the index pages of the B+-Tree to identify the starting page id for the given range. Starting at this page id in the range, and traversing the index pages of the B+-Tree in logical order, we fill up the lookahead buffer **P** with page ids in the range. We then identify contiguous page id sequences in the sorted order of pages in P. Each time the contiguity is violated, we know that the query would have incurred an additional I/O, and we increment the counter appropriately. The buffer is then replenished with page ids so that it is filled again up to the maximum limit *M*.

Our technique above can be viewed as an adaptation of previous work (e.g. [3][4][5]) which used information in the index nodes of a B+-Tree for estimating the *cardinality* of a range predicate. In our case, rather than estimating cardinality, we need to estimate the number of I/Os for scanning that range.

2) *Computing NumIOs*(Q, *Defrag*(I))

Due to the semantics of the defragmentation operation (Section II.D) we know that once an index is defragmented, any range of that index will have no internal or external fragmentation. Thus, the main challenge in estimating *NumIOs*(Q, *Defrag*(I)) is to estimate the number of pages in the range *after* the index has been defragmented. We observe that for this purpose, we can leverage the same technique (based on uniform random sampling) described in Section IV. In this case, we invoke this technique on the pages in the query range. Recall that the estimator thus obtained is an unbiased estimator of the true number of pages in the range after the index is defragmented. The pseudocode for our procedure (called *NumIOsPostDefrag*) is shown in Figure 8. Since *after* the defrag operation, the logical and physical order of pages in the B+-Tree are identical, we know that page ids will be consecutive. Thus each I/O can read in the max number of pages (*M*).

NumIOsPostDefrag(Q, I)
1. Let *r* be the range of the index I scanned by query Q.
2. Sample k% of the data pages in the logical range *r* and use fullness of sampled pages to estimate number of pages in r after it is defragmented (say *N*)
3. Let M = max number of pages read in a single I/O
4. n = N / M
5. **Return** n.

Figure 7. Estimating the number of I/Os for scanning a range of index I if it were to be fully defragmented.

We note that the above two procedures may require adaptations to reflect specific aspects of range scans on a given system (e.g. we have also modeled effect of I/Os due to index pages); but we omit these details here. Instead, we focus on the core ideas of simulating the actual range scan while accessing only the index pages of the B+-Tree; and sampling of leaf pages to estimate internal fragmentation. Finally, similar to the cost model used in query optimizers today, the above procedures do not model transient effects such as buffering on I/Os. In principle, our cost model can be extended by allowing additional parameters that capture such effects. In our experiments (Section VII.C), we evaluate the accuracy and performance of the above "what-if" API in our implementation in Microsoft SQL Server.

B. Estimating Benefit when a Set of Ranges is Defragmented

Let Q be a query that scans a range of index I, and let R be a *set* of non-overlapping ranges of an index I. We will denote by ***Benefit*(Q, I, R)** the reduction in the number of I/Os for query Q if the set of ranges R is defragmented. Note that this generalizes the notion of *Benefit*(Q, I) described in Section A. Consider Figure 8 which shows a query Q and R = {r_1, r_2 and r_3}.

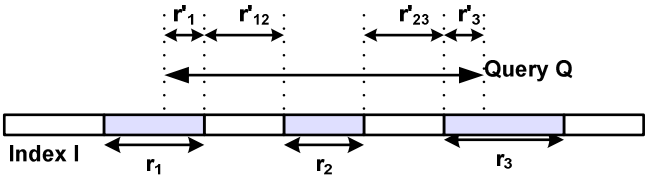

Figure 8. Computing Benefit(Q, I, {r_1,r_2,r_3})

We can express the benefit for query Q using the following equation (Equation 1):

$$Benefit(Q,I,R) = Benefit(r_1',I) + Benefit(r_2',I) + Benefit(r_3',I) + NumIOs(r_{12}',I) + NumIOs(r_{23}',I) - NumIOs(r_{12}',I,R) - NumIOs(r_{23}',I,R)$$

Note that for the ranges of the query Q that overlap with r_1, r_2 and r_3, (respectively r'_1, r'_2 and r'_3), the number of I/Os (if the ranges are defragmented) can be computed as though r'_1, r'_2

and r'_3 are three independent queries and the entire index were defragmented. However, consider the range r'_{12} that is part of Q but is not defragmented. Observe that although the range r'_{12} is not defragmented, its layout can change when R is defragmented. This is because defragmenting a range (e.g. r_1) can require swapping pages. Some of those swaps may involve pages that belong to r'_{12}. Thus, the difference in the number of I/Os for this range after the set of ranges R is defragmented = $NumIOs(r'_{12}, I) - NumIOs(r'_{12}, I, R)$.

There are potentially different ways to model the impact of defragmenting a range r on the layout of another non-overlapping range r'. Below we mention two alternatives, both of which can be modeled at low overhead. In general, the accuracy of such models can be increased by tracking the layout more accurately, but more find-grained models can significantly increase overheads as well.

Independence: The simplest model assumes independence across ranges, i.e. defragmenting r has no impact on the layout of r'. In this case, in Equation 1 above:

$NumIOs(r'_{12}, I) - NumIOs(r'_{12}, I, R) = 0$

$NumIOs(r'_{23}, I) - NumIOs(r'_{23}, I, R) = 0$

Thus, the benefit for the query is modeled as the sum of the benefits for each of the ranges r'_1, r_2 and r'_3.

Uniformity: A more general model of interaction assumes that the impact of defragmenting a range r *uniformly* impacts the layout of all other ranges r'. This implies that larger the range r', the greater is the impact of defragmenting r on it. Furthermore, a conservative model will assume that the impact is adverse, i.e., the number of I/Os for r' will *increase* as a result of defragmenting r. A specific formula based on uniformity and assuming a conservative approach is:

$$NumIOs(r', I, \{r\}) = NumIOs(r', I) + N_{r'} \times EF(r)$$

where *EF(r)* is the measure of external fragmentation (defined in Section II.E), $N_{r'}$ is the number of pages in r', and N is the total number of pages in the index. To extend r to a set of ranges, we would sum up the contributions of each range in the set. Using the uniformity model, Equation 1 above becomes:

$$Benefit(Q, I, R) = Benefit(r'_1, I) + Benefit(r_2, I) + Benefit(r'_3, I) + \frac{(N_{r'_{12}} + N_{r'_{12}})}{N} \sum_{r \in R} EF(r)$$

VI. AUTOMATED RECOMMENDATION OF INDEXES TO DEFRAGMENT

In Section V we introduced "what-if" analysis capability for index defragmentation, i.e. the functionality of being able to estimate the impact of defragmenting an index on the I/O performance of a range scan query. In this section, we describe how such "what-if" analysis capability can be leveraged to develop a client tool that can provide automated recommendations for which indexes to defragment for a given workload. Such a tool can be useful to DBAs, e.g. to help decide which indexes to defragment within a given maintenance batch window.

In Section III.A we outlined the four alternative approaches to index defragmentation: FULL, RANGE, FULL-W, RANGE-W (Figure 2). Recollect that RANGE-W is the most general approach since it uses both workload as well as range-level defragmentation; the other methods do not use one or both of these. In this section, we present an algorithm for the index defragmentation problem (Section III.B). This is an optimization problem which we showed to be NP-Hard via reduction to the Knapsack problem (Appendix A). Note that the algorithm we present below is for the general version of the problem where range-level defragmentation is allowed. Due to the above hardness result, and the similarity of our problem to the Knapsack problem, we use the greedy heuristic for Knapsack in our algorithm below. We have designed our algorithm in such a way that turning off or modifying certain steps in the RANGE-W algorithm will yield solutions for each of the other three alternatives.

A. Algorithm for Recommending Indexes to Defragment

The algorithm for recommending index ranges to defragment for a given workload and defragmentation cost budget (RANGE-W) is shown in Figure 9.

RecommendIndexesToDefragment (**W**, **B**)
W is a workload, **B** is a defragmentation cost budget
1. C = {} // Set of candidate indexes (ranges) to defragment
2. // **Identify candidates based on workload**
3. For each query Q in W
4. For each range scan *r* in Q over an index
5. C = C ∪ *r* // r is the range scanned in query Q
6. // **Identify additional candidates based on data**
7. For each index I referenced in workload
8. Build equi-depth histogram on leading column of index
9. For each bucket *r* in histogram that is accessed by workload
10. C = C ∪ *r* // candidate based on data-only
11. // **Compute total benefit of each candidate index range**
12. For each range *r* in C (say *r* belongs to index I)
13. For each query Q in W that references index I
14. *TotalBenefit*(*r*) += w_Q×*Benefit*(Q, I, {*r*}) // use "what-if" API
15. // **Greedy heuristic for Knapsack**
16. BudgetUsed = 0
17. For each range *r* in C in descending order of *TotalBenefit*(*r*)/*ReorgCost*(*r*)
18. If (budgetUsed + *ReorgCost*(*r*)) ≤ **B**)
19. R = R ∪ *r* ; BudgetUsed += *ReorgCost*(*r*)
20. Update benefit of ranges *r'* ∈ C on same index as *r* // handle interactions
21. **Return** R

Figure 9. Algorithm for recommending index (ranges) to defragment for a given workload and defragmentation cost budget (RANGE-W).

Steps 2-8 are responsible for *candidate* generation, i.e. identifying the space of index ranges to defragment. In particular, we exploit two sources of information for candidate generation. Steps 2-5 identifies as candidates, ranges scanned by queries in the workload. Steps 6-10 introduce additional candidates at a finer granularity than those based on the workload alone. As an example, consider a query that scans an entire index. For this case, these additional candidates we

introduce are important since they can take advantage of range-level index defragmentation. We identify logical ranges of approximately equal width for the ranges accessed in the workload. We use an equi-depth histogram for identifying these ranges. Such a histogram may already be present in the database, and if not, we construct one. Note that approximate equi-depth histograms can be efficiently computed using a uniform random sample of pages of the index (e.g. [6]). Optionally, for efficiency purposes only, we can prune out candidates with very low fragmentation below a threshold.

In Steps 11-14, we compute for each candidate range its total benefit for all queries in the workload (taking into account the weight of each query). This step leverages the "what-if" API described in Section V. Steps 15-20 implement the greedy heuristic for the Knapsack problem. Note that Step 20 needs to be invoked only if the "what-if" API models interactions across ranges within an index (e.g. using the *uniformity* assumption as described in Section V.B). For example if the *independence* assumption is used, then Step 20 can be skipped. The worst case running time of the algorithm in $O(n \times m + m \times \log(m))$, where n is the number of queries in the workload and m is the number of indexes.

We observe that the FULL-W solution can be obtained by modifying: (a) Step 5 in the above algorithm to add the full index instead of range r. (b) Omitting Steps 6-10 that introduce finer grained candidate ranges based on data only. Similarly, to obtain the RANGE algorithm (which does not consider the workload) we: (a) Omit Steps 2-5 (b) Modify Steps 6-11 so as introduce candidates for all indexes rather than those referenced in the workload. (c) Modify Steps 11-14 so that the *TotalBenefit*(r) for a candidate range is Benefit(r, I, $\{r\}$), i.e. the benefit obtained for a range scan on r itself (and not queries in the workload). To obtain FULL, we use the same modifications as in RANGE, except that we also skip introducing candidates based on the histogram and only introduce one candidate per index.

VII. EXPERIMENTS

We have implemented: (1) The range-level index defragmentation API (Section IV); (2) The "what-if" API (Section V) in the storage engine of Microsoft SQL Server 2008; (3) A client tool for recommending which indexes to defragment for a given workload (Section VI). We have implemented all four approaches: FULL, FULL-W, RANGE, RANGE-W. In this section, we present the results of our experimental evaluation on both real world and synthetic databases.

A. Databases and Workloads

We use a real world database and a synthetically generated database for our experiments. The real world database (we will refer to it as REAL) is used by an internal customer, and is used to track business listings. The database is about 11GB; we use the largest table which has around 10 million rows with 16 non-clustered indexes on it. We note that the indexes were already fragmented to different degrees (and we did not alter this in our experiments). For these indexes the external fragmentation varied from around 20% to 90+% and internal fragmentation varied from around 10%-30%.

We also use the synthetic database (SYNTHETIC) since it allows us to systematically vary relevant parameters for our experiments: amount and skew of fragmentation, number of indexes etc. The size of the database varied between 1GB to 10GB depending on the specific experiment.

Finally, we also generated queries in a controlled manner so that we could vary the number of queries in the workload, the width of the ranges (from small ranges to full index scans), the skew in the workload etc. The experiments were run on a Windows Server 2008 machine with 4 processors and 8GB RAM, and an external disk drive. All results reported below are with a cold buffer.

B. Efficiency and Stability of Range-Level Defragmentation

1) Efficiency

To study the efficiency of our range-level index defragmentation method (Section IV.A), we selected several indexes from the real world database. For each index, we logically partition the index into k logical ranges on the leading column of the index, i.e. the partitions together cover the full index. We varied k over the values: 1, 2, 4, 8, 16. Observe that $k = 1$ corresponds to full index defragmentation and $k = 16$ means we defragment each of the 16 ranges using range-level index defragmentation. We measured the total time in each case. We found that the overhead for defragmenting 16 ranges as compared to the full index was between 4%-7%. The overheads for defragmenting for fewer than 16 ranges was even smaller (e.g. for k=4 it is between 2%-5%). This confirms our expectation that our range-level index defragmentation scheme is efficient, i.e., it takes time proportional to the size of the range.

2) Stability

On the REAL database, we verified stability (Section IV.A) of our range-level index defragmentation scheme on several different indexes. For each index we generated an index range (say r) at random and defragmented that range. We then measured the actual number of I/Os required to scan r (say n_1). Next, we selected another range r' on the same index (also at random), and defragmented r'. We again measured the actual number of I/Os required to scan r (say n_2). We compared n_1 with n_2. For each index we repeated this test for different combinations of (r, r'). We found that in almost all cases n_1 and n_2 were either identical. In some cases, they differed by very few I/Os which are due small errors in estimating the *offset* using sampling (Section IV.B). This experiment shows that our scheme satisfies our definition of stability.

C. Evaluation of "What-If" API

In this experiment we evaluate the accuracy of the "what-if" API by comparing the I/Os estimated by the API against the actual number of I/Os. For 4 indexes each on REAL and SYNTHETIC, we generate several different range scan queries by selecting ranges at random (total of 32 queries). For each such query, we estimate the number of I/Os if the index is defragmented using the API in Section V). We then

defragment the index and measure the *actual* number of I/Os for the query. The scatter plot in Figure 10 shows that the estimated number of I/Os vs. actual number of I/Os for each query. The line fit on the chart using linear regression has an R^2 value of 0.98, indicating that our API is able to model the actual number of I/Os reasonably accurately. In particular, this experiment suggests that our API can be used effectively to compare benefits of defragmentation between different range scans.

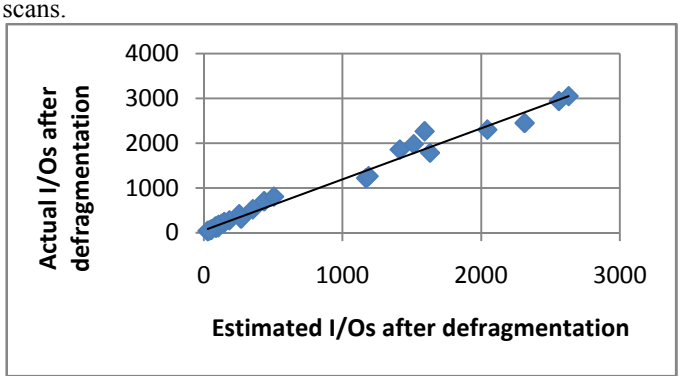

Figure 10. Accuracy of "what-if" API in estimating I/Os after defragmentation.

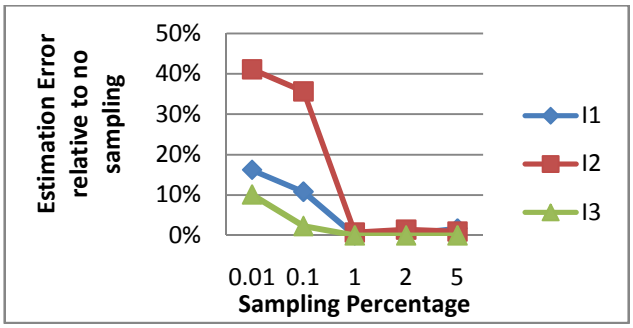

Figure 11. Estimation error for internal fragmentation vs. Sampling percentage

Next, we drill-down into the specific issue of accuracy of estimating the *internal* fragmentation of a given range using uniform random sampling over the leaf pages of the B+-Tree in that range (Section IV.B.2). Recollect that this method is necessary for range-level defragmentation (to estimate the *offset*), as well as for the "what-if" API to estimate the number of I/Os if an index is defragmented. Figure 11 shows that the accuracy of estimating internal fragmentation using sampling improves rapidly as the sampling percentage is increased. In particular, at 1% sampling, the estimation error becomes very small (<1%). This experiment shows that sampling is an effective approach for estimating the internal fragmentation of a range of leaf pages in a B+-Tree.

D. *Comparison of Approaches for Automated Index Defragmentation*

1) *Varying Budget*

For the REAL database, we generate a workload of 8 queries where the ranges are selected at random. We fix the number of indexes scanned by the workload to 4. We vary the budget (measured in units of *ReorgCost* cost model as described in Section III) for defragmenting indexes from 2K units to 100K units and measure the I/O benefit for each method.

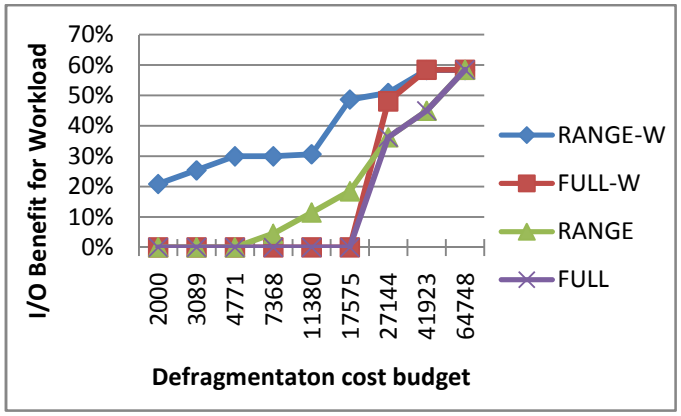

Figure 12. Benefit for workload vs. defragmentation cost budget.

Figure 12 shows that RANGE-W performs the best across all budgets; and in particular is significantly better than all other alternatives for small defragmentation cost budget values. Also, note that the percentage reduction in I/Os is very significant (up to 60%). We see that between RANGE and FULL-W there is no clear winner: as the budget increases FULL-W starts performing better since it is able to select more (full) indexes to defragment which help improve workload performance. In contrast RANGE picks index ranges that have no benefit for the workload. Nevertheless, at lower budgets, RANGE can still be more effective than FULL-W, since some ranges picked do help the workload. Not surprisingly, all techniques perform as well or better than FULL; thus we do not consider FULL in other experiments.

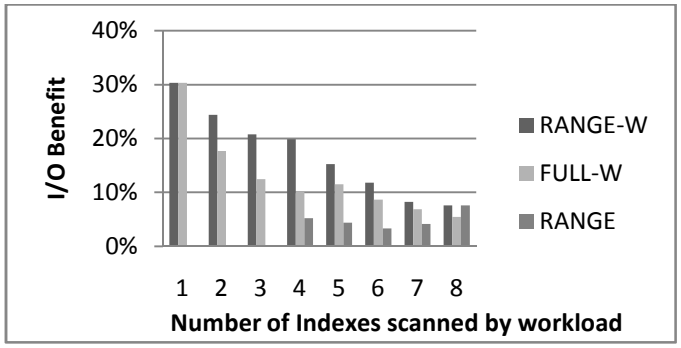

Figure 13. Benefit vs. Number of indexes scanned(fixed defrag budget)

2) *Varying Number of Indexes referenced*

We generated 8 workloads ($W_1..W_8$), where workload W_i performs full scans over i indexes. We fix a small defrag budget (8K units). Thus as the number of indexes referenced increases, the fixed budget becomes smaller as a *percentage* of the total cost of defragmenting all indexes referenced. Thus we expect the % benefits to decrease as number of indexes referenced increases. Figure 13 shows that across all points, RANGE-W gives the largest percentage reduction in I/Os. FULL-W performs better than RANGE when a small subset of indexes (e.g. for W_1) is referenced because it is able to pick

within the small budget the index that is referenced, whereas RANGE selects indexes based purely on fragmentation statistics. However, as the number of indexes referenced increases, the ranges selected by RANGE start to benefit some of the queries in the workload. Since FULL-W can only pick a few *full* indexes to defragment (due to small budget), RANGE does better in some cases (e.g. W_8). This experiment shows that while RANGE-W is the best alternative, both RANGE and FULL-W can be effective in certain cases, and neither is always better than the other.

3) *Importance of Range-Level Index Defragmentation*

In our next experiment, we use the SYNTHETIC database, and generate indexes with the following property: in index I^k a logical range that covers k% (k=30,70, 90) of the pages in the index is injected with a very large amount (EF = 0.99) of external fragmentation. Thus, smaller the k value, the more localized ("skewed") is the fragmentation. For each index, we generate a workload consisting of a single query that scans the entire index.

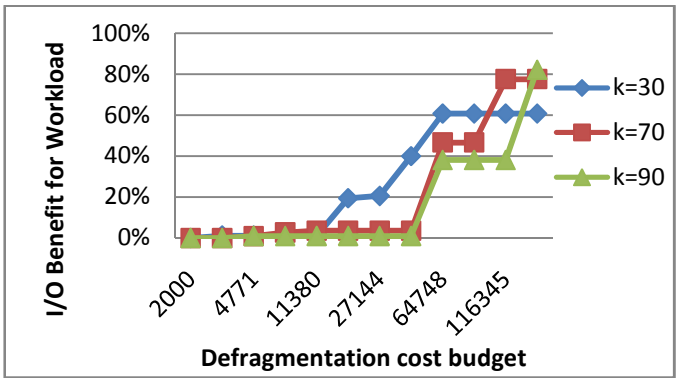

Figure 14. Benefit vs. budget for varying amount of fragmentation

From Figure 14 we see that when the fragmentation is most localized (k=30), the percentage reduction in I/Os for the workload is significant even for low budget values. Note that since the workload is a scan of the full index, in this case the candidates based on workload alone (Steps 2-5 in Figure 9) are not adequate. Thus, this experiment shows that RANGE-W's strategy of identifying candidates based on data (Steps 6-10) helps significantly since it is able to take advantage of range-level index defragmentation. We observe similar results for other values of k as well.

E. *Scalability of RANGE-W Algorithm*

In our final experiment, we evaluate the scalability of RANGE-W algorithm with increasing workload size (on the REAL database). We generate different workloads of size 50, 100, 250, 500 distinct queries; and measure the running time and number of "what-if" API calls issued by the algorithm. We plot these numbers as a ratio relative to the 50 query workload case. We observe from Figure 15 that the running time and number of "what-if" API calls made by RANGE-W both grow sub-linearly with the number of queries. The number of "what-if" API calls increases by a factor of about 8.5 (between 50 and 500 queries), and the running time increases by only a factor of 4. This confirms our expectation that RANGE-W scales well in practice. Finally, we note that even in *absolute* terms the running time is reasonable (e.g. for the 500 query case the running time is 967 seconds, and the number of API calls is 740).

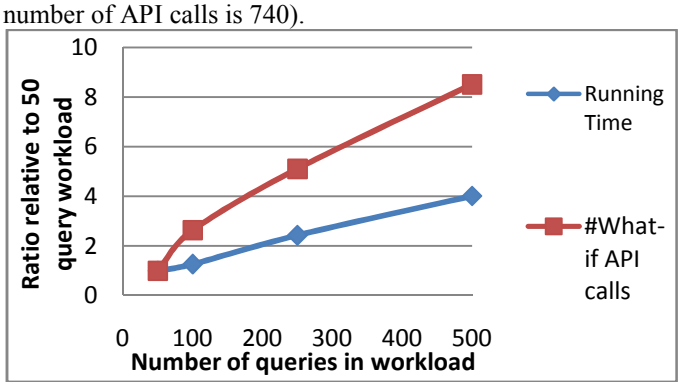

Figure 15. Scalability of RANGE-W with workload size.

VIII. RELATED WORK

The authors of [9] show that fragmentation can have a significant impact on I/O performance of range scan queries, which are common in decision support workloads. These results can be viewed as motivation for our work. All commercial DBMSs support APIs to collect fragmentation statistics about a B+-Tree index including measures of internal and external fragmentation. These are aggregate statistics over the full B+-Tree. They also support the ability to defragment (rebuild and reorganize) an existing B+-Tree. However, we are not aware of any DBMS that supports range-level index defragmentation in a B+-Tree (Section IV). Similarly, no DBMSs today support an API for estimating the impact of defragmentation on I/O performance (Section V).

The state-of-the-art approaches for index defragmentation rely on using fragmentation statistics of an index. As we have shown in this paper, workload driven approaches can perform significantly better. We have also shown that using range level defragmentation can greatly increase effectiveness.

The idea of leveraging the workload for performance tuning problems such as physical database design (e.g. [1][9][13]), statistics selection (e.g. [7][10]), histogram refinement and cardinality estimation (e.g. [1][13]) etc. has been studied previously. However, techniques for studying how to exploit workload information for the index defragmentation problem have not been studied previously.

Our technique for estimating the *number of I/Os* for a range scan query over a B+-Tree index (Section V) leverages information in the index pages of the B+-Tree. This technique can be viewed as an adaptation of previous work (e.g. [3][4][5]) which used information in the index nodes of a B+-Tree for estimating the *cardinality* of a range predicate.

IX. CONCLUSION AND FUTURE WORK

We have presented novel techniques relating to index defragmentation: a "what-if" analysis API in the database engine for estimating the impact of defragmenting an index on I/O performance; range-level index defragmentation; formalization and hardness of the index defragmentation

problem, and a scalable algorithm for the problem. Our experiments on real and synthetic databases in Microsoft SQL Server demonstrate the promise of our techniques.

In this paper we have adopted an *offline* model where the DBA needs to specify a workload and decide when to implement the recommendations. The problem of *online* index defragmentation, i.e. "closing the loop" automatically, is an interesting area of future work. Finally, the question of how index fragmentation affects performance for other forms of stable storage besides disks also remains unanswered.

ACKNOWLEDGMENT

We thank Nico Bruno, Surajit Chaudhuri, Raghav Kaushik, Christian König, David Lomet and Ravi Ramamurthy for valuable feedback.

APPENDIX A

We show hardness for a special case of the Index Defragmentation problem (defined in Section III.B), where: (a) *ReorgCost*(I) = N_l, the number of pages in index I. (b) The reduction in I/O cost is obtained by eliminating internal fragmentation only, i.e.,(there is no external fragmentation) . Thus *Benefit*(Q, I) = (1 – *CR*(I)). *NumPagesScanned*(Q, I), where *CR(I)* (0 < *CR(I)* ≤ 1) is the compaction ratio for index I, and *NumPagesScanned*(Q, I) is the number of pages of index I scanned by query Q.

Claim: The Index Defragmentation Problem is NP-Hard.

Proof: By reduction from the Knapsack problem.

Knapsack problem: Given a set of *n* items S, where each element s ∈ S has *Value*(*s*), *Weight*(s), and a knapsack size B; pick a subset K ⊆ S such that $\sum_{s\in K}$ *Weight*(*s*) ≤ B and $\sum_{s\in K}$ Value(s) is maximized.

Given an instance of the Knapsack Problem, we generate an instance of the Index Defragmentation Problem as follows. For each item s ∈ S, we create a table with a single column, and we define a clustered index on that column. We also generate one query per item *s*, which performs a full scan of the clustered index. We insert data into the table such that the number of pages in the index is *Weight*(*s*). We ensure that the compaction ratio *C* of the index and weight *w* of the query that scans the index, are set such that the following equation is satisfied.

$$Value(s) = w \times (1 - CR) \times Weight(s)$$

In general there are multiple solutions of *w* and *CR* that can satisfy the above equation. One such solution is as follows: If the ratio $\frac{Value(s)}{Weight(s)} \leq 1$, we set *w* = 1and *CR* = $(1 - \frac{Value(s)}{Weight(s)})$. If $\frac{Value(s)}{Weight(s)} > 1$, we set $w = 2 \times \left\lceil \frac{Value(s)}{Weight(s)} \right\rceil$, and $CR = \frac{Value(s)}{Weight(s) \times 2 \times \left\lceil \frac{Value(s)}{Weight(s)} \right\rceil}$. Observe that in both cases we ensure 0 < *CR* ≤ 1.

By the above construction, the defragmentation cost of the index I_s corresponding to item *s* = NumPages(I_s) = *Weight*(*s*). Note also that the *Benefit*(Q_s, I_s) = *w*.(1 – *CR*). *NumPagesScanned*(Q_s, I_s) = *w*. (1-*CR*) . NumPages(I_s) = *Value*(*s*).

Thus, solving the above instance of the Index Defragmentation problem with a defragmentation budget **B** is a solution to an arbitrary instance of the Knapsack problem.

APPENDIX B

Lemma: For any range level index defragmentation scheme $\mathfrak{I}$ that satisfies the equivalence and efficiency properties to be stable, when defragmenting any range *r* on index I, it must use an offset equal to the number of pages in *p(r)* in the fully defragmented index.

Proof: We show this by using an adversary argument. The adversary constructs an index where there is external fragmentation but no internal fragmentation. Defragmenting requires only reordering pages but no compaction. In this case, the number of pages in the index is the same before and after defragmentation (let this value be N). The adversary first picks a range r_1 for defragmentation such that $|p(r_1)| = |s(r_1)| = k$; i.e. there are *k* pages in the prefix as well as suffix of range *r*. Observe that due to the above construction, *k* is the number of pages in *p(r) after* I is defragmented as well. Now, suppose the scheme $\mathfrak{I}$ uses offset *k'* (≠ *k*) when defragmenting *r*. If *k'*<*k*, the adversary chooses the next range to defragment as *p(r)*. Since there *k* pages in *p(r)* are at least one page belonging to *r* is in the first *k* pages of the B+-Tree, this means that at least one page of *r* will be swapped out of its location when *p(r)* is defragmented. This violates stability since because is $\mathfrak{I}$ efficient, it cannot correct such fragmentation by reorganizing *r*. Similarly, if *k'*>*k,* the adversary chooses *s*(*r*) to defragment, and once again this violates stability.

REFERENCES

[1] Aboulnaga A., Chaudhuri S. *Self-Tuning Histograms: Building Histograms without Looking at Data*. In SIGMOD 1999.

[2] S. Agrawal et al. *Database Tuning Advisor for Microsoft SQL Server*. In Proceedings of VLDB 2004.

[3] M. J. Anderson et.al. *Index Key Range Estimator*. U. S. Patent 4,774,657, IBM Corp., Armonk, NY, Sep. 1988. Filed June 6, 1986.

[4] G. Antoshenkov. *Random sampling from pseudo-ranked B+ trees*. In Proceedings of the 19th Very Large Data Bases, pages 375–382. Morgan Kaufmann, 1992.

[5] P.M. Aoki. *Generalizing "Search" in Generalized Search Tree*. In IEEE ICDE, 1998.

[6] S. Chaudhuri, R. Motwani, V.Narasayya. *Random Sampling for Histogram Construction: How much in enough?* In SIGMOD 1998.

[7] S. Chaudhuri, V. Narasayya. *Automating Statistics Management for Query Optimizers*. In Proceedings of ICDE 2000.

[8] W.G. Cochran. *Sampling Techniques*. John Wiley & Sons, New York, third edition, 1977.

[9] B. Dageville et al. *Automatic SQL Tuning in Oracle 10g*. In Proceedings of VLDB 2004.

[10] A. El-Helw et al. *Collecting and Maintaining Just-in-Time Statistics*. In Proceedings of ICDE 2007.

[11] M.R. Garey, and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

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

[13] M. Stillger et al. *LEO – DB2'sLEarning Optimizer*. In VLDB 2001.

[14] Zilio et al. *DB2 Design Advisor: Integrated Automated Physical Database Design*. In Proceedings of VLDB 2004.