Partitioning Techniques for Fine-grained Indexing

Eugene Wu, Samuel Madden
CSAIL, MIT
eugenewu@mit.edu
madden@csail.mit.edu

Abstract— **Many data-intensive websites use databases that grow much faster than the rate that users access the data. Such growing datasets lead to ever-increasing space and performance overheads for maintaining and accessing indexes. Furthermore, there is often considerable skew with popular users and recent data accessed much more frequently. These observations led us to design Shinobi, a system which uses horizontal partitioning as a mechanism for improving query performance to cluster the physical data, and increasing insert performance by only indexing data that is frequently accessed. We present database design algorithms that optimally partition tables, drop indexes from partitions that are infrequently queried, and maintain these partitions as workloads change. We show a $60\times$ performance improvement over traditionally indexed tables using a real-world query workload derived from a traffic monitoring application**

I. Introduction

Indexes are the standard method for improving the performance of selective queries and the past decade has seen considerable research focused on selecting a near-optimal set of indexes for a representative workload [1]. A careful balance must be maintained between creating too many indexes, which sacrifices disk space and insert performance, and creating too few indexes, which results in poor query performance. Furthermore, as the indexes grow alongside the datasets, the performance and resource costs can be very high for a number of reasons. First, updating the index for rapidly arriving data can be very expensive; for example, we found that installing a single varchar attribute index on a 3.4 GB table in Postgres or MySQL can reduce insert performance by up to $40\times$. Second, the total index size can easily rival that of the dataset – a snapshot of Wikipedia's *revision* table from 2008 uses indexes that total 27 GB for 33 GB of raw data that does not include article text. In order to constrain the amount of index space, index selection tools require a maximum space bound [1]. Third, online reoptimization by creating and dropping indexes on large, unpartitioned tables is prohibitively expensive.

Our key observation about many workloads is that despite rapidly growing data sizes, the amount of accessed data increases at a far slower pace. For example, Cartel [2] is a sensor-based system we built for collecting data from cars as they drive around Boston. The *centroidlocations* table stores GPS information of participating cars every second and has grown to over 18 GB in a few years. Yet the workload only accesses 5% of the table on any given day, and more than 50% of the queries access data from just the last day. Similar query skew exists for Wikipedia's *revision* table, which stores metadata information of every article's revision history. 99.9% of the requests access the 10% of records that represent the most recent revision of an article.

If the queries always access a small subset of the table, then a clear optimization is to split the table into the queried and non-queried partitions, and *selectively index* the partitions that are beneficial. Many applications already do this – warehouses may partition the fact table into historical and recent transactions and only index the latter. Unfortunately, the policies to define the partitions and decide which partitions to index have so far been adhoc, or have not taken the tradeoff of query performance and index updates into account.

Additionally, data is not always clustered on the keys the table is partitioned on. For example, a workload consisting of spatial queries will benefit from partitioning *centroidlocations* by the lat, lon attributes; however, the records are not likely to be physically ordered by their lat, lon values, which leads to excessive disk seeks when answering the queries [3]. Range partitioning the data along the keys will group records with similar values together and reduce the number of disk seeks.

In this paper, we describe Shinobi, a system that uses partitioning to provide fine-grained indexing and improves the performance of skewed query workloads, while optimizing for index update costs. Shinobi uses three key ideas: first, it partitions tables, such that regions of the table that are frequently queried together are stored together, separate from regions that are infrequently queried. Second, it selectively indexes these regions, creating indexes on partitions that are queried frequently, and omitting indexes for regions that are updated but queried infrequently. Third, over time, it dynamically adjusts the partitions and indexes to account for changes in the workload. Shinobi takes as input a set of indexes, a set of keys to partition on, a query workload, and machine statistics such as RAM and the table size, and uses a cost-based partitioner to find the optimal range partitioning of the table and the best set of indexes for each partition. As the workload evolves, Shinobi minimizes the amount of repartitioning necessary to re-optimize the system for the new workload characteristics. Shinobi is intended for workloads with predicates on ordered attributes (e.g., salary or time). In other workloads, it is sometimes possible induce an ordering on the queried attributes to utilize Shinobi's optimizations [4].

Our contributions toward partitioning in a single-machine database are as follows:

1) *Enabling selective indexing with partitioning*. Shinobi chooses the optimal partitions to index, which dramatically reduces the amount of data that is indexed. In our

experiments using a workload from Cartel, Shinobi can avoid indexing over 90% of the table and reduce index update costs by 30× as compared to a fully indexed table without sacrificing performance.

2) *Partitioning based clustering*. Shinobi optimally partitions tables for a given workload, which increases query performance by physically co-locating similarly queried data. Using the same Cartel workload, we improve query performance by more than 90× as compared to an unpartitioned, fully indexed, table.
3) *Reducing index creation costs*. Shinobi only indexes partitions that are frequently accessed. By splitting the table into smaller partitions, the cost of creating an index on a single partition becomes cheaper, which lets the system make fine-grained optimizations.
4) *Novel workload lifetime estimation*. Shinobi uses a novel online algorithm that uses past queries to estimate the number of queries the workload will continuously access in a given data region.

II. Related Work

There is a large body of related work in the areas of automated index selection and partitioning, index optimization, adaptive databases and partial indexes.

Database Designers. Modern database design tools use query optimizer extensions to perform *what if* analysis [5] – at a high level, the optimizer accepts hypothetical table configurations and queries as input and outputs the optimizer estimates. The optimizer's wealth of statistics and its highly tuned cost model are powerful tools for estimating the cost of a potential workload. Shinobi uses a cost model that does not attempt to replicate decades of optimizer research [6], [7], but rather identifies a small set of parameters for evaluating various table configurations on a mixed query and insert workload.

Index selection tools explore the space of potential indexes and materialized views. Both offline [1], [8], [9] and online [10], [11], [12] tools find an optimal set of indexes within user specified constraints (e.g., maximum index size). Rather than replicate this work, Shinobi analyzes the output of such tools (or hand-crafted physical designs), and runs index selection and partitioning techniques to further optimize their designs by identifying subsets of a table where installing an index will be detrimental to performance.

Partitioning techniques such as [3], [13], [14], [15] partition tables using workload statistics in order to improve query performance. However, they do not explicitly consider index update costs during cost estimation. In contrast, Shinobi accounts for both query and insertion costs and uses partitioning as a mechanism for dropping indexes on infrequently queried portions of the data.

Optimized B-Trees. To optimize B-tree insert performance, most work focuses on minimizing insert overheads by buffering and writing updates in large chunks. Such work include insert optimized B-trees [16], [17], [18], and Partitioned B-trees [19], [20], for traditional disk based systems, and flash optimized B-trees such as [21]. Shinobi is agnostic to any particular indexing technique as it focuses on *dropping indexes* on partitions where indexes are not beneficial. Regardless of the index that is being used, we can still realize insert performance wins on insert intensive workloads.

Adaptive Storage. Database Cracking [22], [23] and other adaptive indexing techniques incrementally sort and index the underlying table based on the query workload. It creates a copy of the keyed column and incrementally sorts the column as a side effect of normal query execution. Partial-sideways cracking is an extension that only replicates the queried data ranges rather than the entire column. Database cracking is intended for in-memory databases and has been shown to perform comparably to a clustered index without the need to provide a set of indexes up front. Adaptive Indexing [24] is similar in spirit and leverages partitioned B-trees for block-oriented (e.g., disk) storage. However, it fully indexes the table and would still benefit from dropping indexes from unqueried data ranges.

Partial Indexes. Finally, partial indexes [25] are a method for building an unclustered index on a predicate-defined subset of a table. Seshadri and Swami [26] propose a heuristic-based method that uses statistical information to build partial indexes given a constraint on the total index size. Unfortunately, there are several practical limitations to partial indexes. First, in all partial index implementations we know of, the query optimizer only uses a partial index when it can determine that queries access a strict subset of the index; by physically partitioning a table and creating conventional indexes on a subset of partitions, we avoid this subset limitation. Second, partial indexes cannot be clustered because multiple partial indexes can overlap; this limits the applicability of partial indexes to all but the most selective queries. In contrast, Shinobi can cluster indexes just like in a conventional system. When we used Postgres' partial indexes for the experiments in Section VI-A, each query on average took 20 seconds to execute while index creation took nearly 2000 seconds. On the other hand, Shinobi can partition and index the same data in 500 seconds and execute the same queries in 0.1-0.8 seconds on average. Thus, one way to view our work is as an efficient implementation of clustered, non-overlapping partial indexes.

III. Architecture

Shinobi partitions and indexes tables to efficiently process workloads with a high insert-to-query ratio. The input to Shinobi is a list of attributes each table is to be partitioned on, a set of indexes to install on the table, and a set of queries and inserts that apply to the table. Indexes may be provided by a database administrator or database tuner (e.g., [27]). Shinobi finds an optimal set of non-overlapping range partitions and chooses indexes for each partition (together denoted as the *table configuration*) to maximize workload performance.

Shinobi supports arbitrary queries over SQL partitions. Most DBMSs support the ability to store a table in partitions and direct queries over a partitioned table to the appropriate partitions (in our implementation we use the master/child

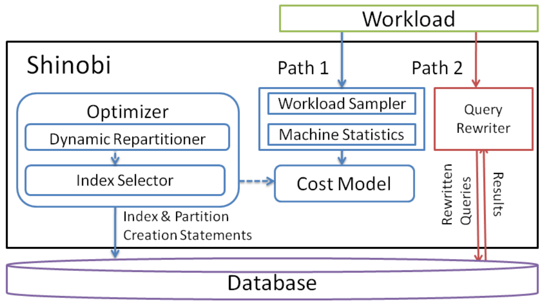


Fig. 1. The Shinobi architecture

partitioning feature of Postgres [28]; MySQL includes similar features).

Shinobi acts as an intermediary between a database and the workload. It consumes a workload and outputs rewritten queries and inserts as well as SQL to repartition and re-index the table. Shinobi can be used both to find an initial, optimal table configuration for a static workload and to continuously optimize the configuration under a dynamically changing workload.

Figure 1 illustrates the system architecture. The solid and dashed arrows indicate the query/data and call paths, respectively. The workload follows two paths. Path 1 samples incoming SQL statements and updates workload statistics for the *Cost Model*. The *Optimizer* uses the cost model to (re)optimize the table configuration. Path 2 parses queries using the *Query Rewriter*, which routes queries with predicates on the partitioning attribute to the relavent partitions. Queries without such predicates are directed to all partitions.

The *Workload Sampler* reads recent SQL statements from the query stream and computes workload characteristics such as the insert to query ratio, and the query intensity of different regions of the table. Similarly, the *Machine Statistics* component estimates capabilities of the physical device as well as database performance information. Physical statistics include RAM size and disk performance while database statistics include append costs, insert costs, and typical query costs (see Table IV-B for a full parameter list.)

The *Cost Model* uses these statistics to calculate the expected statement cost for a workload. The key idea is that the model takes into account not only query cost but also the non-trivial cost of updating indexes on inserts and updates. The *Index Selector* and *Dynamic Repartitioner* components both use the *Cost Model* to optimize the table configuration. The *Index Selector* calculates the best set of indexes to install on each partition of a table and the *Dynamic Repartitioner* re-optimizes the table configuration as the workload varies and calls the *Index Selector* to decide which indexes to build.

IV. Cost Model

In this section, we introduce models for predicting the average cost per query in a workload, the cost to repartition and reindex a table, and the overall benefit of switching to a new table configuration. These models are used in Section V to choose the optimal index configuration and partitioning.

Our cost model estimates the cost of range scans over single tables (though the system itself can handle any query). We preprocess the queries fed into our optimizers to extract a set of ranges that they access from each table. Key-foreign key joins between a table T_1 with primary key k and a table T_2 with foreign key fk referencing k are treated as a range scan on k in T_1 and a range scan on fk in T_2 with value restrictions on k or fk propagated from the other table (if any such restrictions exist.) Joins without such value restrictions are treated as complete scans of all partitions of the underlying table (as such joins are likely to be executed via hash or sort-merge joins which scan tables in their entirety.) Our current preprocessor is somewhat limited and will discard complex queries which it cannot analyze; we are currently able to handle all of the queries issued against the CarTel database we use for evaluation, but implementing a more sophisticated preprocessor is an area for future work.

The goal of the cost model is to accurately order the query and update performance of different table configurations, and not to exactly estimate the expected cost of all types of queries. As our experiments validate, the simplified cost model is enough to achieve this goal and allow us to see large performance gains.

A. Variables

The values of the model constants were derived experimentally and are shown in Table IV-B. Additionally, the following is a list of common variables (and their values measured on a 3.4 GB database running Postgres 8.1) used throughout the rest of this paper. To improve readability, we assume that W and I are globally defined and available to all cost functions and algorithms.

$W = W_q \cup W_i$: The workload W consists of a set of select queries W_q and insert statements W_i over a single table.

$\Pi = \{p_1, .., p_N\}$: The partitioning Π is composed of N range partitions over the table. Each partition is defined by a set of boundaries, one for each of D dimensions $p_i = \{(s_{d,p_i}, e_{d,p_i}] | d \in \{1, .., D\}\}$.

$I = \{I_1, .., I_m\}$: The predetermined set of m indexes to install on the table (from a database administrator, for instance).

$\Psi = \{\psi_i \subseteq I | 0 \leq i \leq N\}$: The set of indexes to install on each partition. ψ_i defines the set of indexes to install on partition p_i. Ψ and its corresponding partitioning Π always have the same number of elements.

B. Query Cost Model

The query cost model estimates the average expected cost per statement in W given Π and Ψ. To a first approximation, the average statement cost is proportional to a combination of the average *select* and *insert* cost.

$$cost(\Pi, \Psi) \sim a * cost_{select} + b * cost_{insert}$$

We use the probabilities of a select and insert statement for a and b, respectively,

$$cost(\Pi, \Psi) = \frac{|W_q|}{|W|} \times cost_{select}(\Pi, \Psi) + \frac{|W_i|}{|W|} \times cost_{insert}(\Psi)$$

TABLE I

MODEL STATISTICS AND THE VALUES USED IN EXPERIMENTS

RAM	512MB	Amount of memory
data size	3400MB	size of the table
$cost_{seek}$	5ms	disk seek cost
$cost_{read}$	18ms/MB	disk read rate
$cost_{dbcopy}$	55ms/MB	write rate within PostgreSQL
$cost_{createindex}$	52ms/MB	bulk index creation rate
$icost_{fixed}$	0.3ms	record insert cost (no index updates)
$icost_{overhead}$	.003ms/MB (.019ms/MB)	insert overhead per MB of indexes clustered (unclustered) data
$lifetime_W$	variable	Expected # queries in workload W

We now consider how to evaluate $cost_{select}$ and $cost_{insert}$.

C. Select Costs

The main components that determine select cost are the cost of index and sequential scans over each partition. We make the simplifying assumption that a query q uses the index in ψ_P that can serve its most selective predicates, and the cost is proportional to the amount of data being accessed. Additionally, we consider the cases where the heap file is physically ordered on the partitioning key (*clustered*), and when it is not (*unclustered*).

The model considers the select cost of each partition separately, and calculates the weighted sum as the select cost across the entire table:

$$cost_{select}(\Pi,\Psi) = \sum_{p,\psi_p \in \Pi,\Psi} \frac{|W_q \cap p|}{|W_q|} \times cost_{pselect}(W_q \cap p, p, \psi_p)$$

Where $W_q \cap p$ is the set of queries that access p, and $cost_{pselect}()$ is:

$$cost_{pselect}(W_{qp}, p, \psi_p) = \sum_{q \in W_{qp}} \frac{\begin{cases} iscan(\frac{|q \cap p|}{|p|}, p) & \text{q uses } \psi_p \\ seqscan(p) & otherwise \end{cases}}{|W_{qp}|}$$

$cost_{pselect}$ is the average cost per query in W_{qp}. $seqscan$ is the cost of a sequential scan, and modeled as the sum of the seek cost plus the cost of reading the partition:

$$seqscan(p) = cost_{seek} + size(p) \times cost_{read}$$

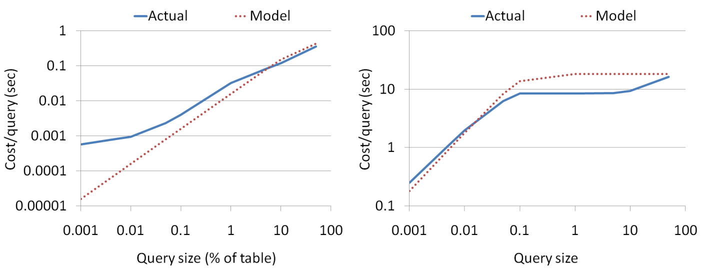
where $size(p)$ is the size in MB of p.

$iscan$ is the cost of scanning an index and depends on whether the data is clustered. If it is, then the cost is modeled as a disk seek plus a sequential scan of the query result:

$$iscan(s,p) = cost_{seek} + s \times size(p) \times cost_{read} \qquad \text{data is clustered}$$

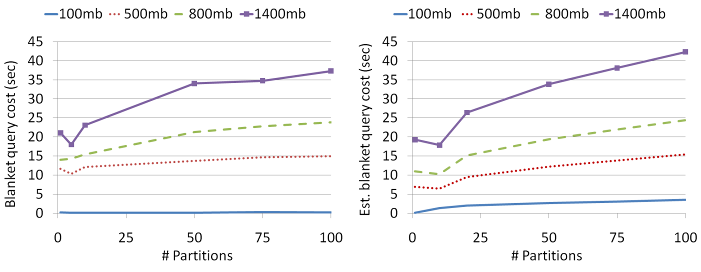
However if the data is not clustered, the cost is dependent on the query selectivity, s, and the size of the partition, p, w.r.t. the size of RAM. It is modeled using a sigmoid function that converges to the cost of a sequential scan [29]. We assume that the database system is using bitmap scans that sort the page ids before accessing the heap file [30]. In this case, for scans of just a few records, each record will be on a different heap-file page; as more records are accessed, the probability of several records being on one page increases. Eventually, all pages are accessed and the cost is identical to a sequential scan. The speed that the function converges to its maximum is dependent on a parameter k which depends on the size of the table and whether or not it fits into memory. We experimentally measured k to be 150 when the partition fits into RAM, and 1950 when it does not:

$$iscan(s,p) = seqscan(p) \times \frac{1-e^{-k \times s}}{1+e^{-k \times s}} \qquad \text{data not clustered}$$



(a) size(table) $<$ RAM (b) size(table) $>=$ RAM

Fig. 2. Query cost w.r.t. query selectivity



(a) Real cost (b) Estimated cost

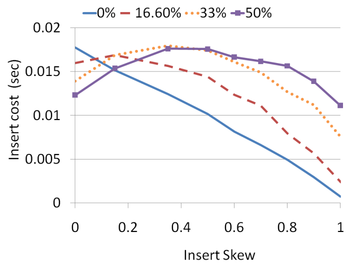
Fig. 3. Blanket query cost for varying table sizes (curves) and # partitions (x-axis)

Figure 2 compares the actual and model estimated costs of queries using an unclustered index on a machine with 512 MB of memory for two different table sizes – one much smaller than physical memory (155 MB) and one much larger (996 MB). The selectivities vary from 0.001% to 100% and each query accesses a random range. In Figure 2(a), the model under-estimates the cost for very small queries and over-estimates the cost for queries larger than .1% in Figure 2(b), however the overall shapes are similar. We found the curves to be consistent for smaller and larger table sizes, although the cost curves for when the tables are very close to the size of memory lie somewhere in-between.

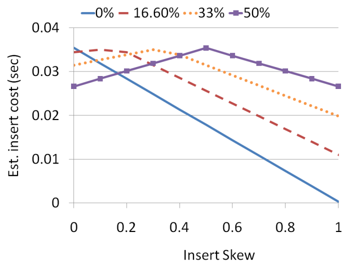
Queries that don't contain a predicate on the partitioning key (*blanket queries*) must execute the query on all of the partitions and combine the results. A blanket query incurs $cost_{pselect}$ on every partition (Figure 3). We believe the slight "dip" in query cost is because each partition becomes small enough to fit into memory, thus switching the $cost_{pselect}$ curve towards the curve in Figure 2(a).

D. Insert Costs

The average cost of an insertion into a partitioned table is dependent on the total size of all indexes, and the distribution of inserts across the various partitions. For simplicity, we assume that the distribution of inserts within a partition is uniform, whereas there may be skew across partitions. Although this can overestimate the insert cost for large partitions, the accuracy improves as partitions are split. We first describe how



(a) Actual costs



(b) Model estimates

Fig. 4. Insert cost w.r.t. fraction of data in smaller table (curves) and insert skew (x-axis)

to model the cost of inserting into a single partition, followed by a model for multiple partitions.

1) Single Partition: The insert cost of a single partition, π_i, is modeled as the sum of a fixed cost to append the record to the table, $icost_{fixed}$, and the overhead of updating the indexes (e.g., splitting/merging pages, etc) installed on the partition. We experimentally observed that this cost is linearly proportional to the size of the index. The overhead is the product of the cost of updating each MB of index, $icost_{overhead}$, and the total size of all indexes on the partition in MB:

$$cost_{insert}(\psi_i) = icost_{fixed} + icost_{overhead} \times \sum_{u \in \psi_i} size(u)$$

where $size(u)$ is the size in MB of index u. $size(u)$ can be easily calculated from the sizes of the partition keys and the number of records in the partition.

It is widely known that B-tree insertions take time proportional to $log_d(N)$, where d is the fan-out and N is the number of records in the tree [31]. Our experiments showed that PostgreSQL insertion costs increase linearly rather than logarithmically as the total size of the indexes grows, which is surprising. We believe the reason why update performance deteriorates given larger total index sizes is that with larger tables, each insert causes more dirty pages to enter the buffer pool, leading to more evictions and subsequent page writes to disk. [32] and experiments on Oracle observed similar behavior.

2) Two Partitions: For simplicity, we first describe the model for varying insert distributions between two partitions, π_0 and π_1, and their respective sets of indexes ψ_0 and ψ_1. Intuitively, the insert cost will be maximized when the insertions are distributed uniformly across the ranges of both partitions (analogous to a single table of size=$size(\{p_0\}) + size(\{p_1\})$); conversely, the cost will be minimized when all of the inserts are directed to p_0 or p_1. As described above, the cost of an insertion is directly proportional to the sizes of the installed indexes. The insert cost can be modeled with respect to an *effective total index size ($size_{et}(\psi_0, \psi_1)$)* that varies in size based on the insert distribution:

$$cost_{insert}(\psi_0, \psi_1) = icost_{fixed} + icost_{overhead} \times size_{et}(\psi_0, \psi_i)$$

$size_{et}$ is modeled using a modified triangle function where its value at the peak is the total size of ψ_0 and ψ_1 whereas the minimums are equal to the size of either ψ_0 or ψ_1:

$$totalsize = size(\psi_0) + size(\psi_1)$$
$$size_{et}(\psi_0, \psi_1) = totalsize - \sum_{j=0,1} max\left(0, \left(size(\psi_j) - totalsize * \frac{|W_i \cap \psi_j|}{|W_i|}\right)\right)$$

where $\frac{|W_i \cap \pi_j|}{|W_i|}$ is the percentage of the insert workload that inserts into partition π_j.

Figure 4 compares the actual and model estimated costs of inserts with varying data and insert skew on a machine with 512 MB of memory. We used a single 600 MB table that is split into two partitions; the size of the smaller partition varies between 0% to 50% of the original table ($curves$). The distribution of inserts within each partition is uniform, however the percentage of inserts into the small partition ($x - axis$) varies from 0% to 100%. For each partition configuration ($curve$), the insert cost is most expensive when the distribution is uniform across the dataset – when the smaller partition contains 25% of the data, the insert cost is maximized when it serves 25% of the inserts. Although there is a nonlinear component to the cost, our model captures the overall trend very well.

3) N Partitions: The above model naturally extends to N partitions, Π, and the respective indexes, Ψ. $size_{et}(\Psi)$ is modeled by a multidimensional triangle function:

$$totalsize = \sum_{\psi_k \in \Psi} size(\psi_k)$$
$$size_{et}(\Psi) = totalsize - \sum_{\psi_j \in \Psi} max\left(0, \left(size(\psi_j) - totalsize * \frac{|W_i \cap \psi_j|}{|W_i|}\right)\right)$$

E. Repartitioning Cost Model

The repartitioning cost model estimates the cost to switch from one table configuration to another. It takes as input the existing configuration Π_{old}, Ψ_{old} and the new configuration Π_{new}, Ψ_{new}, and calculates the cost of creating the new partitions and indexes. We measured the cost of dropping existing partitions or indexes to be negligible. This repartitioning cost is used in the partition optimizers to balance repartitioning costs against improved workload performance. For clarity, we use $\bullet$ to denote the arguments (Π_{old}, Ψ_{old}, Π_{new}, Ψ_{new}).

1) Partition Costs: The total partitioning cost, $repart_{part}$, is the sum of the cost of creating the new partitions:

$$repart_{part}(\bullet) = \sum_{p \in \Pi_{new}} createp(p, \{(p_i, \psi_i) \in (\Pi_{old}, \Psi_{old}) | p_i \cap p \neq \emptyset \wedge p_i \neq p\})$$
$$createp(p, \Lambda_\cap) = \sum_{p_\cap, \psi_\cap \in \Lambda_\cap} (cost_{pselect}(W_{create, p\cap}, p_\cap, \psi_\cap) + \frac{size(|p_\cap \cap p|)}{cost_{dbcopy}})$$

The second argument to $createp$ is the set of existing partitions and indexes that intersect the new partition p. If the new partition already exists, there is no need to create it, and the argument will be the empty set. $createp$ is the cost of creating p; it is the aggregate cost of querying each intersecting partition, $p_\cap$, for the new partition's data and writing the data into p (at $cost_{dbcopy}$ MB/sec). $W_{create,p\cap}$ is the workload consisting of queries that select data belonging in p.

2) Indexing Costs: The cost of installing indexes is directly proportional to the size of the partition being indexed:

$$repart_{idx}(\bullet) = \sum_{(p,\psi)\in(\Pi_{new},\Psi_{new})} createindex(p, \psi, \Pi_{old}, \Psi_{old})$$

$createindex$ is the cost of creating the indexes ψ for p. It is modeled as the product of p's size, the cost to index one MB of data and the number of indexes to create:

$$createindex(p, \psi, \Pi_{old}, \Psi_{old}) = size(p) \times cost_{createidx} \times \\ |\psi \setminus \{x \in \psi_j | p_j = p \wedge (p_j, \psi_j) \in (\Pi_{old}, \Psi_{old})\}|$$

Note that if p already exists and has indexes installed, the cost of recreating them is not included in the cost.

3) Total Cost: Given the previous partitioning and indexing models, the total repartitioning cost is the sum of $repart_{part}$ and $repart_{idx}$:

$$repart(\bullet) = repart_{part}(\bullet) + repart_{idx}(\bullet)$$

F. Workload Cost Model

The workload cost model calculates the expected benefit of a new table configuration over an existing configuration across the new workload's lifetime.

$$benefit_W(\bullet) = (cost(\Pi_{old}, \Psi_{old}) - cost(\Pi_{new}, \Psi_{new})) * lifetime_W$$

$lifetime_W$ is the expected lifetime, in number of queries, of the new workload before the workload shifts to access a different set of data. This value is useful for the *Dynamic Repartitioner* in order to estimate the total benefit of a new table configuration and balance it against the cost of repartitioning the table. As the value increases, the partitioning cost is amortized across the workload so that more expensive repartitioning can be justified. This value can be calculated as the sum of the lifetimes of the query only workload, $lifetime_{W_q}$, and the insert only workload, $lifetime_{W_i}$.

$$lifetime_W = lifetime_{W_q} + lifetime_{W_i}$$

In Section V-C, we present an online algorithm that learns the expected lifetime of a query-only or insert-only workload and test its effectiveness in Section VI-A.3.

G. Total Workload Benefit

The total benefit of a new configuration, $benefit_{total}$, including repartitioning costs, is defined as:

$$benefit_{total}(\bullet) = benefit_W(\bullet) - repart(\bullet)$$

V. Optimizers

This section describes Shinobi's three primary optimizers that use the cost model to partition the table, select indexes for each partition, and repartition the table when the workload changes, and a strategy for estimating the value of $lifetime_W$. We begin with by describing the *Index Selector* as it is needed by the repartitioner.

A. Index Selector

The goal of *Index Selector* is to find the Ψ that minimizes the expected cost workload W on a database with partitions Π. Formally, the optimization goal is:

$$\Psi_{opt} = \underset{\Psi}{\text{argmin}}(cost(\Pi, \Psi))$$

Finding the naive solution to this optimization problem requires an exhaustive search ($O(2^{|\Pi|*|I|})$) because the indexes do not independently affect the cost model Instead, we use a greedy approach that adds k indexes at a time, stopping once a local maximum is reached. The parameter k dictates how thoroughly to explore the search space. When $k = |\Pi||I|$, the algorithm is equivalent to an exhaustive search. In our experiments k is set to 1 which reduces the runtime to $O((|\Pi||I|)^2)$. This algorithm is very similar to Configuration Enumeration in [9], which sets $k = 2$ in the first iteration, then uses $k = 1$ in subsequent iterations.

B. Dynamic Repartitioner

The Dynamic Repartitioner merges, splits and reindexes the partitions as the workload evolves and existing table configurations become suboptimal. For instance, if the workload shifts to a large, unindexed partition, the cost of sequentially scanning the partition will be very high, while creating an index reduces insert performance; the Dynamic Repartitioner will split the partition so that the queried ranges are isolated. In order to avoid costly repartitions that marginally improve workload performance, this component uses $benefit_{total}$ (section IV-F) to evaluate whether a new configuration is worth the repartitioning cost.

We use an N-dimensional quad-tree (where N is the number of partitioning attributes) that splits/merges partitions if the query performance is expected to improve. Each leaf node represents a single partition containing a sub-range of the data. The tree implements the method $getPartitions()$, which returns the partitioning represented by the leaf nodes.

Algorithm 1 takes as input the tree representation of the current partitioning ($root$) and the current indexing (Ψ), and outputs an optimized logical partitioning (no data is moved while the algorithm runs) that the optimizer uses to physically partition the data. Reoptimization begins with a merge phase followed by a split phase; each phase takes $root$ and Ψ as input and returns the root of the modified tree. The order of the phases is not important [1]. The merge and split algorithms are nearly identical, so we present them together and highlight the differences in italics.

The goal of the merging [*splitting*] phase (Algorithm 1) is to find the set of nodes to merge [*split*] that will maximize the expected benefit (as defined in IV-F) over the existing partitioning. Π is used to estimate the benefit of candidate partitionings and $benefit_{best}$ tracks the benefit of the best partitioning so far (lines 1,2). In each iteration of the **while** loop, $nodes$ is initialized with the parents of the leaf nodes

[1] If the nodes can have a variable number of children (e.g., a node can have 2, 3, or 4 children), then it is necessary to merge prior to splitting so that the tree can transform into any configuration.

[*all of the leaf nodes*] (line 4). The algorithm searches for the node to merge [*split*] that will maximize the benefit over $benefit_{best}$ (lines 6-15). This is done by temporarily merging [*splitting*] the node (line 7) in order to calculate the benefit of the new partitioning (lines 8-10), and then reverting to the previous tree (line 11). If a node that increases $benefit_{best}$ is not found, the algorithm returns the root of the tree (line 17). Otherwise the node is merged [*split*] and $benefit_{best}$ is updated to the benefit of the new partitioning (lines 19-20).

The runtime of the merge algorithm is limited by the number of leaf nodes, and the fan-out. For L nodes and a fan-out of F, the algorithm may run for L/F iterations in order to merge L/F nodes, and call $SelectIndex$ with lookahead=1 on L/F nodes in each iteration, for a total runtime of $O((L/F)^2(L|I|)^2)$. The split algorithm can theoretically run until every partition contains a single record, but can be bounded by setting a minimum allowable partition size.

In our experience, splitting occurs far more frequently than merging. The only reason to merge is if the overhead of extra seeks becomes significant relative to the cost of accessing the data. For example, if the workload switches to an OLAP workload consisting of large scans of the table, then the optimizer will consider merging partitions.

```
1: $\Pi \leftarrow$ root.getPartitions()
2: $\text{benefit}_{\text{best}} \leftarrow 0$
3: while true do
4:    nodes $\leftarrow \{$l.parent$|$l $\in$ root.leaves()$\}$ [root.leaves()]
5:    benefit, node $\leftarrow$ 0, null
6:    for n $\in$ nodes do
7:       n.merge() [n.split()]
8:       $\Pi' \leftarrow$ root.getPartitions()
9:       $\Psi' \leftarrow$ SelectIndex($\Pi'$,1)
10:      benefit$'$ = benefit($\Pi, \Psi, \Pi', \Psi'$)
11:      n.split() [n.merge()]
12:      if benefit$'$ > benefit $\wedge$ benefit$'$ > $\text{benefit}_{\text{best}}$ then
13:         benefit, node $\leftarrow$ benefit$'$, n
14:      end if
15:   end for
16:   if node = null then
17:      return root
18:   end if
19:   node.merge() [node.split()]
20:   $\text{benefit}_{\text{best}} \leftarrow$ benefit
21: end while
```

Algorithm 1: MergePartitions/SplitPartitions(root, Ψ) [*Differences in italics*]

C. Estimating Workload Lifetime

As we noted earlier, $benefit_{total}$ is highly dependent on the value of $lifetime_W$, defined as the number of SQL statements for which the workload will continue to access (read or write) approximately the same data range. This section describes an algorithm that estimates the lifetime of a workload by sampling the SQL statements.

The high level idea is to split the table into M equal sized ranges and keep track of the lifetime of each individually. For each range, we store a vector of lifetime values, where a lifetime consists of a number of timesteps during which at least one query accessed (read or write) the range. The most recent lifetime increases until the range is not queried for several timesteps, whereupon a fresh lifetime value is appended to the vector. The lifetime of a given range is computed as a weighted moving average of the individual lifetimes in the vector. The lifetime of a partition is calculated as the average lifetime of the intersecting ranges. We now describe the details below.

For ease of explanation, we focus on a single range r_i. We describe how to 1) update its lifetime vector $v_i = [lt_1, .., lt_N]$ and 2) derive r_i's lifetime value. lt_1 and lt_N are the lifetimes of the oldest and most recent lifetime in the vector, respectively.

The naive approach for updating v_i is as follows: during each time interval, if range r_i is queried at least once, then lt_N is incremented by one. Otherwise a new lifetime (lt_{N+1}) is added to v_i by appending 0. To avoid over-penalizing if r_i is not queried for many timesteps, we only append to v_i if lt_N is nonzero. The drawback of this approach is that it keeps no history, so it is completely dependent on current workload conditions. For example, if r_i is consistently accessed every other timestep, the lifetime will be reset every other timestep and the range will never have a chance to be partitioned.

In light of this, we use an additional count variable c_i, which maintains an estimate of the number of queries that have accessed r_i in the past. In each timestep, c_i is first multiplied by a decay factor, $\alpha \in [0, 1]$, which controls the number of future timesteps a query is counted, and then incremented by the number of queries that access r_i in the current time interval. During a given timestep, lt_N is incremented by 1 if $c_i > \tau$; otherwise a new lifetime is added to v_i as in the naive approach.

Finally, r_i's lifetime is calculated as the exponentially weighted average of the values in v_i, where β is the decay factor. In our experiments, we derived $\alpha = 0.2$, $\tau = 0.01$, and $\beta = 0.2$ by simulating a sample workload using the cost model and running a greedy algorithm for each factor.

VI. EXPERIMENTS

In the following subsections, we describe experiments that show the utility of Shinobi for partitioning and indexing tables and the resulting space savings and performance gains.

Our current prototype is written in Python and issues SQL commands to a backend database (this work used PostgreSQL and MySQL). Each partition is implemented as a separate table, and queries are rewritten to execute on the partitions. A partition is created by executing a ``create table as select...'' SQL query that reads the relevant data from the existing partitions and adds the data to the new partition table. The experiments use a dual 3.2 GHz Pentium IV with 512 MB of RAM and a 300GB 7200 RPM drive, running Redhat Linux 2.6.16, PostgreSQL 8.1.10 and MySQL 5.0.27. Shinobi uses SQL transactions to repartition tables, quiescing the system until reorganization is complete.

Although Shinobi uses SQL transactions to repartition the tables, we quiesce the system until repartitioning is complete so that partitioning and workload costs can be clearly distinguished.

A. Multi-Dimensional Cartel Workload

In this section, we run Shinobi as an end-to-end system on a two-dimensional dataset and several workloads. After describing the experimental dataset and workload, we first show that Shinobi can reduce the total cost of this workload by over an order of magnitude. Then, we illustrate the utility of the adaptive lifetime estimator, and finally explain how partitioning can be used to avoid re-indexing costs on workloads that exhibit cyclic properties. Due to space limitations, we do not include results for one-dimensional experiments, however the results are very similar. Note that Shinobi works well for, but is not limited to, spatial datasets – any workload that queries tables via ordered attributes can benefit from our techniques.

1) Dataset, Workload and Approaches: The dataset is the *centroidlocations* table consisting of lat, lon, timestamp, and several other identification attributes. The values of the lat and lon fields are approximately uniformly distributed within the ranges $[35, 45]$ and $[-80, -70]$ (the Boston area), respectively, which we define as the dataset boundary. The table size is 3.4 GB, contains 30 million records, and is partitioned and indexed (unclustered) on the lat, lon composite key.

In the following experiments, we use a realistic Cartel workload, W_{cartel}, and two synthetic workloads, $W_{lifetime}$ and W_{cyclic}. All workloads contain multiple timesteps; each timestep contains a number of spatial range queries followed by a large number of insertions uniformly distributed across the table. The queries access 0.1% of the table in a square spatial bound.

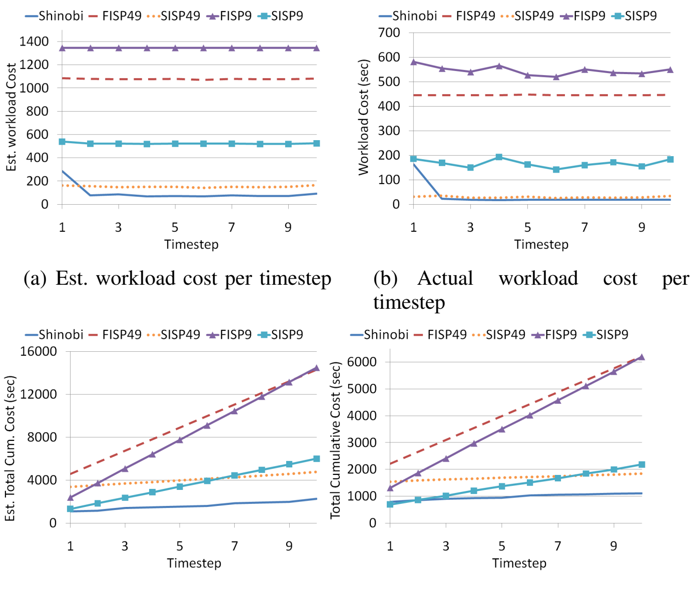
The Cartel workload (W_{cartel}) contains 10 timesteps and uses queries generated from the Cartel database's daily trace files between November 19, 2009 and December 5, 2009. To generate the queries in a timestep, we pick a trace file, compute the distribution of data that the file accesses, and sample 100 queries from the distribution. We then generate 360 inserts for each query (36k/timestep), which is the ratio we found when processing the trace files.

The first synthetic workload ($W_{lifetime}$) contains 10 timesteps and showcases how Shinobi responds to skewed workloads that shift the "hot spot" at varying intervals. Each timestep generates queries from a gaussian distribution (σ=5$\times$ query size) centered around a random lat,lon coordinate. On average, each workload uniquely accesses about 8% of the table. $W_{lifetime}$ has the same number of queries in each timestep, however the number of queries vary between 1 and 1000, depending on the experiment – more queries in each timestep means the workload accesses the same data for a longer time and thus simulates a less dynamic workload than one that accesses different data very often. The ratio of inserts to queries is fixed at 100 inserts per query.

The second sythetic workload (W_{cyclic}) contains 8 timesteps, where each timestep is generated in the same way as in $W_{lifetime}$. The center point of the gaussian repeats after every 3 timesteps – the repetition helps illustrate the cumulative cost of creating new indexes each time the workload moves. Figure 8 visualizes two of the distributions.

We compare approaches that differ along two dimensions: index selection technique and partitioning type. Full Indexing (FI) indexes all of the data in the table, and Selective Indexing (SI) uses the algorithm described in Section V-A to only create beneficial indexes. Static Partitioning (SP_N) partitions the table into N equally sized partitions, and Optimized Partitioning (OP) finds the optimal partitioning as described in Section V-B.

The approaches are a fully indexed table ($FISI_1$); full and selective indexing on a table statically partitioned into N partitions ($FISP_N, SISP_N$); and selective indexing on a dynamically partitioned table ($SIOP$ or $Shinobi$).



(a) Est. workload cost per timestep (b) Actual workload cost per timestep

(c) Est. cumulative workload cost (d) Actual cumulative workload cost

Fig. 5. Shinobi performance on Cartel 2D workload

2) Cartel Results: In this experiment we run Shinobi on a realistic workload (W_{cartel}) to validate the accuracy of the cost model. We find that Shinobi performs as well as the best statically partitioned configuration and avoids the high initial cost of fully partitioning and indexing the table. The goal is to maximize total system performance, so the optimizers also take (re)partitioning costs into account.

Figure 5(b) shows the workload only performance over the 10 timesteps. Although not graphed, $FISP_1$ took on average 1100 sec per timestep. The $FISP_{9,49}$ curves illustrate the effectiveness of statically partitioning the table into 9 and 49 partitions, respectively. Increasing the number of partitions from 1 to 9 and 49 reduces the select query costs by over 3$\times$ and 4$\times$, respectively. Selective indexing only creates indexes on heavily queried partitions, and reduces insert costs for $SISP_{9,49}$ by 7$\times$ and 21$\times$, respectively. In fact, for timesteps 5-7, $SISP_{49}$ didn't create any indexes. Shinobi performs as well as $SISP_{49}$; the higher initial cost is because the estimated $lifetime_W$ is still small, so that Shinobi uses a non-optimal but much cheaper partitioning. As $lifetime_W$ increases, Shinobi further partitions the table so that Shinobi performs very close to $SISP_{49}$ by timestep 2 and slightly out-performs $SISP_{49}$ by timestep 3. Overall, Shinobi out-performs $FISP_1$ by over 60×.

Figure 5(d) plots the cumulative cost of partitioning the table and running the workloads. For reference, $FISP_1$ took 11,000s to run the experiment. The values in timestep 1 are dominated by the initial partitioning and indexing costs. Splitting the table into 9 and 49 partitions costs 660 and 2500s, respectively, while indexing all of the data costs 240s. Although selective indexing ($SISP_{9,49}$) can avoid indexing a large fraction of the partitions and reduce indexing costs by almost 200s, these initial partitioning costs are still substantial. The reason for such high costs is because each partition is created by a query that accesses the partition's contents via an index scan of the full table. In contrast, Shinobi chooses a cheap partitioning because the estimated $lifetime_W$ is still low, and creates new partitions by accessing existing partitions.

The slopes of the curves represent the workload performance and any repartitioning or indexing costs. $FISP_9$ and $SISP_9$ have a low initial cost, but quickly outpace $FISP_{49}$ and $SISP_{49}$, respectively, due to higher query costs when accessing larger partitions. However, it is interesting to note that $SISP_9$ out-performs the more optimally partitioned $FISP_{49}$ simply by reducing index update costs. Shinobi converges to the same slope as $SISP_{49}$ and initially partitions the table in 2.5× less time. The slope between timesteps 1 and 5 are slightly higher because of additional repartitioning costs that are justified by an increasing $lifetime_W$ value. *Shinobi*'s total repartitioning costs are lower than that of $FISP_{49}$ and $SISP_{49}$ because the cost of splitting a partition becomes significantly cheaper as the partitions become smaller, and because only the queried data regions, rather than the full table is partitioned. Most importantly, *Shinobi processes the entire workload before $FISP_{9,49}$ and $SISP_{49}$ finish processing the first timestep and out-performs $FISP_1$ by 10×.*

Figures 5(a) and 5(c) validate the accuracy of the cost model. Although Shinobi scales the expected costs up, we preserve the relative differences between the different strategies. For example, we correctly predict the cross-over point between Shinobi and $SISP_{49}$ in Figure 5(a).

To verify that the performance trends observed are not specific to PostgreSQL, we ran an identical experiment using a MySQL-MyISAM backend and found similar trends, with 49 partitions performing better than 1 or 9 partitions, and with selective indexing significantly reducing insertion costs.

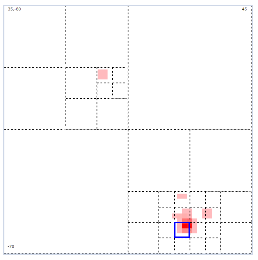


Fig. 6. Shinobi's 2D partitions and indexes after timestep 5. Dotted boxes are partitions, solid edged boxes are indexed partitions, filled boxes are queries (more queries results in darker fill).

Figure 6 shows the resulting table configuration after timestep 5 on W_{cartel}; Shinobi focuses partition costs on regions that are heavily queried. The filled boxes are queried regions – more queries result in a darker fill; the dotted boxes are partitions and the solid blue edged boxes (e.g., in the lower right quadrant) are indexed partitions.

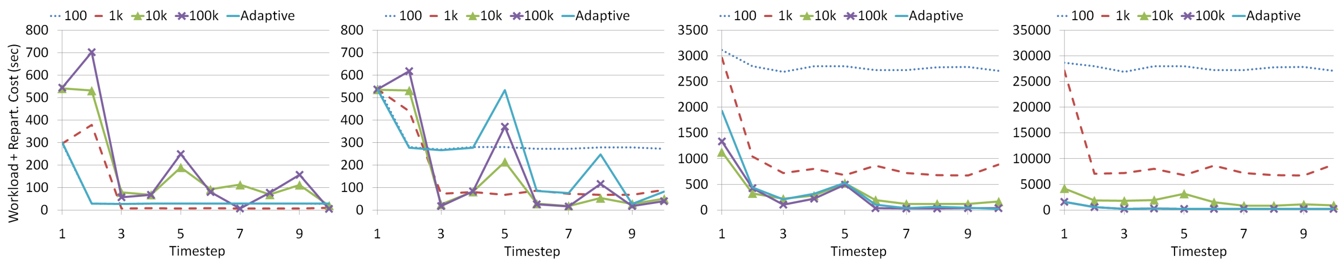
3) Lifetime Estimation: In this set of experiments, we analyze Shinobi's adaptivity to workloads ($W_{lifetime}$) that access different regions with varying rates of dynamism and show the importance of accurately predicting the value of $lifetime_W$. We show that Shinobi running with the adaptive $lifetime_W$ estimator performs comparably to a "lookahead" that knows the number of queries in a timestep prior to executing it (Shinobi must adaptively estimate it). The lookahead is configured with static $lifetime_W$ values ranging from 100 to 100k queries.

In each timestep, the lookahead approaches load the new workload, run the repartitioning algorithm using the given $lifetime_W$ value, and execute the workload to completion. On the other hand, the adaptive approach estimates the new $lifetime_W$ in an online fashion.

Figure 7 shows the workload plus repartitioning costs at each timestep when the workload lifetime is 100, 1k, 10k and 100k SQL statements. We find that for most cases, a naive lookahead algorithm that sets $lifetime_W$ to the actual length of the workload results in the best performing curve. However, this does not always occur, as in Figure 7(c), where the 100k curve outperforms the 10k curve. The reason is that the naive approach disregards the fact that two consecutive workloads may overlap, and therefore underestimates $lifetime_W$ for shorter workloads. In general if the workload is long running, it is better to over-estimate $lifetime_W$ and over-partition the table, rather than to run every query sub-optimally. Shinobi always splits the table into 4 partitions in the first timestep because it reduces the select costs from 60 to 20 seconds.

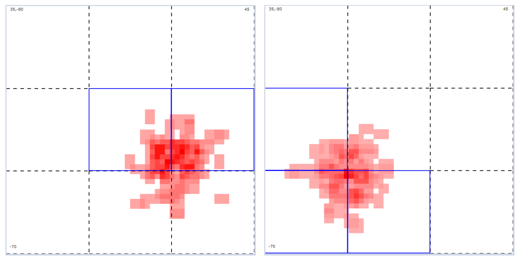
The adaptive $lifetime_W$ estimator ($Adaptive$) performs competitively in all of the experiments. In Figure 7(a), its curve is nearly identical to the 10k curve and in the other experiments, it converges to the optimal curve. The cost of the adaptive algorithm is the start-up time; it needs to wait for enough samples before the $lifetime_W$ matches the actual workload lifetime and the optimizer decides to re-optimize the table layout. During this period, the query performance can be suboptimal and Shinobi may repartition the same set of data several times. This is clear in Figure 7(b), where $Adaptive$ closely resembles the 100 curve in the first 4 timesteps. In timesteps 5 and 8, the $lifetime_W$ value in the queried region is large enough that Shinobi decides to repartition the table, thus reducing the workload cost in subsequent timesteps.

4) Reindexing Costs: Although selective indexing alone can improve insert performance and drastically reduce the amount of data that must be indexed, it still incurs a high reindexing overhead for workloads that cycle between multiple hot spots because it creates indexes for the current workload only to drop the indexes soon after the workload moves. For example, Figure 8 depicts the workload in two of the timesteps. Timestep 1 (Figure 8(a)) indexes two of the nine



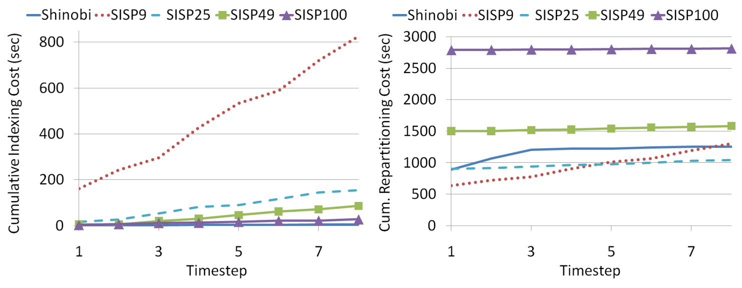
(a) Actual Lifetime = 100 (b) Actual Lifetime = 1k (c) Actual Lifetime = 10k (d) Actual Lifetime = 100k

Fig. 7. Shinobi performance with static and adaptive $lifetime_W$ values (curves) for different actual lifetimes (plots)



(a) Timesteps 1, 4, and 7 (b) Timesteps 2, 5, and 8

Fig. 8. Visualization of W_{cyclic}.



(a) Index costs (b) Index and partitioning costs

Fig. 9. Repartitioning overheads of different partition schemes on W_{cyclic}

partitions. These indexes are dropped when the workload shifts away in timestep 2 (Figure 8(b)), only to be recreated in timestep 4.

Partitioning is an effective way of alleviating re-indexing overheads. First, smaller partitions allow the dataset to be indexed at a finer granularity, while also reducing the cost of indexing a particular partition. Second, sequential scan performance may be fast enough that indexes are not needed for marginally accessed partitions. This trend is clear in Figure 9(a), which plots the cumulative indexing costs over the 8 timesteps in W_{cyclic}. Increasing the number of partitions decreases the slope, because fewer indexes are created. Shinobi does not incur any indexing overhead because it creates small enough partitions that scanning the partitions is fast. As a comparison, Figure 9(b) plots the sum of partitioning and indexing costs. The $SISP$ approaches must trade off between indexing costs and partitioning the entire table before running the workload. Shinobi partitions the table during the first three timesteps, then incurs no overhead for the rest of the experiment.

Finally, we note that even with the recurring indexing costs, selective indexing still out-performs fully indexing or not indexing the entire table.

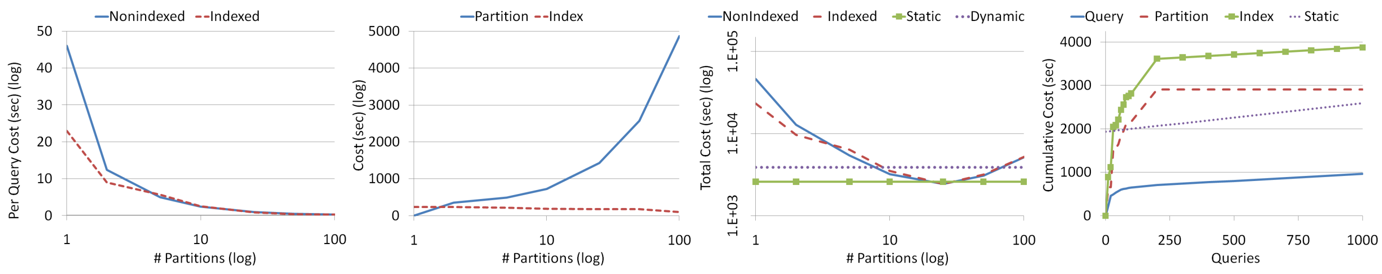
B. Partitioning Experiments

This section compares different partitioning policies to understand the query benefits of partitioning (without selective indexing) on an unclustered dataset. We use the same dataset as above but values of the timestamp column are uniformly distributed throughout the table. The table is indexed and partitioned on the timestamp attribute. This set of experiments only considers a non-indexed or fully indexed table. The workload contains 1000 select queries that each accesses a random 0.1% range of the table via a predicate on the timestamp colmun, and no insertions. Figure 10 shows the results.

The first experiment partitions the table into an increasing number of equal sized partitions and then executes the workload to completion. The cost per query decreases inversely with the number of partitions (Figure 10(a)). Postgres executes each index scan via a bitmap scan which sorts the ids of the pages containing relevant records, and reads the pages in order. Since the records are not clustered, the index must still read a large number of pages to retrieve them (bounded by the partition size). Beyond 5 partitions, the cost of scanning the partition converges to the cost of accessing the data using an index. Increasing the query selectivity shifts the convergence point to the right.

As expected, the cost of partitioning increases linearly with the number of partitions. Interestingly, the total indexing cost slightly *decreases* (Figure 10(b)). This is because an index grows as $Nlog(N)$ where N is the size of the table. Additionally, there is improved cache locality as more of the partition and the index nodes can fit into memory.

Figure 10(c) plots the sum cost of partitioning, indexing, and running the workload. As in Figure 10(a), indexing is most beneficial when there are less than 5 partitions, above which the cost of creating the indexes outweighs its query benefit. The optimal number of partitions (25 for this workload) shifts to the left (right) for shorter (longer) workloads. Section VI-A.3 analyzes Shinobi's online algorithm for estimating a workload's lifetime. The $Static$ curve is the result of Shinobi reading the entire workload a priori and finding the optimal quad-tree based partitioning (32 partitions). For comparison, the $Dynamic$ curve is the total cost when optimizing the table layout using Shinobi's online algorithms.



(a) Query performance (b) Partitioning costs (c) Sum of all costs (d) Shinobi's online optimizer

Fig. 10. Query impact and overhead costs as the number of partitions varies.

Figure 10(d) depicts the total costs of the dynamic repartitioner (Section V-B, uses a stacked graph to plot the query, (query + partitioning) and (query + partitioning + indexing) costs after processing the N^{th} workload query. The optimizer is run every 10 queries and Shinobi only repartitions the table if the expected future query benefit outweighs the partitioning cost. The $Static$ curve is included as reference. At the beginning of the workload, Shinobi is penalized with expensive queries before aggressively partitioning the table in nearly every round until the 200^{th} query. At the end, the online algorithm is within 1.5× of the static optimizer.

Finally, we describe how Shinobi responds to increasing query sizes and blanket queries, however limit ourselves to a high level description due to space constraints. In general, Shinobi tends to favor splitting over merging – merging partitions is an expensive operation and only reduces the seek overhead of accessing multiple partitions. This overhead is small compared to the size of large or blanket queries, thus only very small partitions will be merged. Shinobi may even split partitions if the queries partially overlap with them in order to minimize the sizes of queried partitions. Blanket queries are treated in a similar fashion, and the partitioning is optimized for the non-blanket queries. In fact, Figure 3 illustrates that a small number of partitions can even improve the performance of blanket queries.

C. Selective Indexing Experiments

In this section, we use a clustered version of the CarTel dataset, and show how the workload performance and the size of the indexes change as a function of varying workload characteristics. Although a clustered dataset is highly optimized for query performance (in fact, partitioning does not improve query performance at all), we show that Shinobi's selective index can still significantly improve performance by reducing insert overheads.

Because insert performance is directly related to the size of the indexes, we report the percentage of the table that is indexed ($\%indexed$) and the expected cost per SELECT statement ($Select$). The workload consists of 100 queries generated from an exponentially decaying distribution over the timestamp values and a varying number of inserts uniformly distributed throughout the table. By generating synthetic queries, we are able to control a range of system parameters, including a) *Query Size*, the percentage of the table each query accesses (Default: 1%), b) *Insert to Query Ratio (IQR)*, the number of insert statements for every select statement (Default: 100), c) *# Partitions*, the number of equally sized partitions the table is split into (Default: 20), and d) *Partitions Accessed (PA)*, the number of partitions that the workload accesses (Default: 9). Figure 11 shows how the $Select$ and $\%indexed$ curves vary with respect to the above characteristics. The left Y-axis displays the percentage of the table indexed, from 0% to 100%. The right Y-axis shows the values of the $Select$ curve, in seconds.

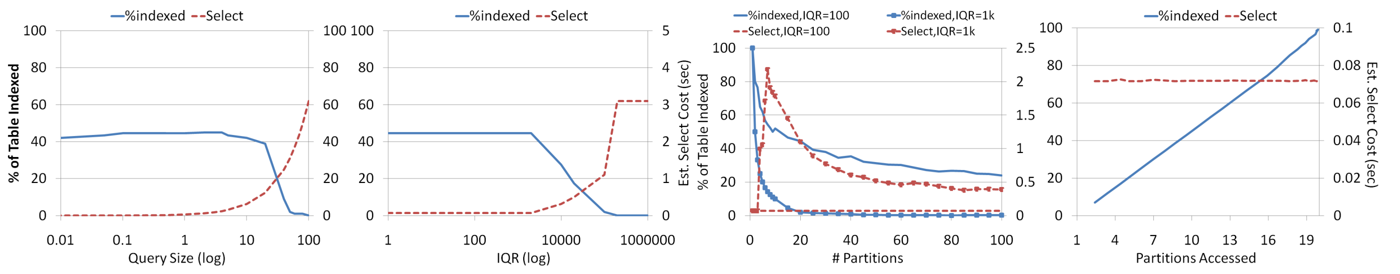
Figure 11(a) varies the query size from 0.01% to 100% of the table, plotted on a logarithmic scale. Shinobi indexes all of the queried partitions when the queries are smaller than 5% (the size of a partition). When the query size exceeds 5%, Shinobi chooses not to index fully read partitions. Above 75%, the cost of maintaining indexes exceeds their query benefit and all indexes are dropped. The "cliff" in the curve shifts to the left as inserts become more expensive (e.g., data is unclustered). As expected, $Select$ cost increases as more data is accessed.

Figure 11(b) varies the IQR from 1 to 100k, also plotted on a log scale. The $\%indexed$ curve starts at 45%, where all of the queried partitions are indexed, and starts decreasing past IQR=2000 because the indexes on sparsely queried partitions are dropped. Shinobi continues to drop indexes until IQR = 100K, at which point none of the partitions are indexed. Naturally, the $Select$ curve increases as more queries are executed with sequential scans of the partitions. However this is justified when insert costs become the dominant factor.

Figure 11(c) show that partitioning is a good mechanism for fine grained indexing without impacting query performance. We show two pairs of curves for IQR=100 (no markers) and IQR=1k (with markers). When the IQR is low, Shinobi indexes all accessed partitions so that the $Select$ curve stays flat (the curve is near the x-axis), however this limits the number of partitions that do not need indexing. For a larger IQR, when the workload is insert-dominated, the $Select$ curve increases as Shinobi aggressively drops indexes on sparsely queried partitions, then gradually decreases as partition sizes decrease.

Figure 11(d) varies the number of partitions accessed by spreading out the queries. As expected, the amount of indexed data grows with the number of accessed partitions while the $Select$ cost stays constant. This shows that Shinobi is most effective when the workload accesses a small subset of the data.

Reducing the size of the index also reduces insert overhead.



(a) Vary query size (b) Vary insert-query ratio (c) Vary number of partitions (d) Vary number of accessed partitions

Fig. 11. Percentage of table that is indexed and avg. statement cost as various workload characteristics change

Eliminating a full index can reduce insert costs by 3-40×, depending on whether or not the data is clustered. By dropping indexes on large subsets of the table, Shinobi can drastically reduce insert costs, even for clustered datasets.

D. *Optimization Runtime Overhead*

We also ran a number of experiments to measure the runtime of the optimization algorithms themselves. We omit the details due to space constraints, but we found that for all of the experiments above, calling the repartitioner cost less than 1 second (far less than the actual repartitioning time.) The dominant cost is choosing the partitions, which grows quadratically with the number of partitions and amount of table accessed, and in pathological cases can grow to take several seconds when there are hundreds of partitions. These costs are still likely much less than the actual repartitioning times.

VII. Conclusions

This paper presented Shinobi, a system that horizontally partitions and indexes databases for *skewed query workloads* containing queries that access specific regions of the data (which may vary over time) and possibly many inserts spread across large portions of the table. Our key idea is to partition the database into non-overlapping regions, and then selectively index just the partitions that are accessed by queries. We presented an index-aware cost model that is able to predict the total cost of a mix of insert and range queries, as well as algorithms to select and dynamically adjust partitions and indexes over time and reorder records so that popular records are close together on disk.

Our experiments show partitioning significantly reduces query costs when the dataset is not clustered on the partition keys, whereas selective indexing can dramatically reduce the index size, and correspondingly the index costs, even for clustered datasets. We show dramatic performance improvements on a real-world two-dimensional query workload from a traffic analysis website, with average performance that is 60× better than an unpartitioned, fully indexed database.

References

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya, "Automated selection of materialized views and indexes for sql databases," in VLDB, 2000.
[2] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden, "CarTel: A Distributed Mobile Sensor Computing System," in SenSys, 2006.
[3] P. Cudre-Mauroux, E. Wu, and S. Madden, "Trajstore: An adaptive storage system for very large trajectory data sets," in ICDE, 2010.
[4] E. Wu, "Shinobi: Insert-aware partitioning and indexing techniques for skewed database workloads," Master's thesis, MIT, 2010.
[5] S. Chaudhuri and V. Narasayya, "Autoadmin "what-if" index analysis utility," in SIGMOD, 1998.
[6] M. Jarke and J. Koch, "Query optimization in database systems," in ACM Computing Surveys, 1984.
[7] S. Chaudhuri, "An overview of query optimization in relational systems," in PODS, 1998.
[8] G. Valentin, M. Zuliani, and D. C. Zilio, "Db2 advisor: An optimizer smart enough to recommend its own indexes," in ICDE, 2000.
[9] S. Chaudhuri and V. Narasayya, "An efficient, cost-driven index selection tool for microsoft sql server," in VLDB, 1997.
[10] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in ICDE, 2007.
[11] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-line index selection for shifting workloads," in SMDB, 2007.
[12] K.-U. Sattler, M. Luhring, I. Geist, and E. Schallehn, "Autonomous management of soft indexes," in SMDB, 2007.
[13] S. Ceri, M. Negri, and G. Pelagatti, "Horizontal data partitioning in database design," in SIGMOD, 1982.
[14] S. Papadomanolakis and A. Ailamaki, "Autopart: Automating schema design for large scientific databases using data partitioning," in SSDBM, 2004.
[15] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in SIGMOD, 2004.
[16] G. Graefe, "Write-optimized b-trees," in VLDB, 2004.
[17] C. Jermaine, "A novel index supporting high volume data warehouse insertions," in VLDB, 1999.
[18] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," Acta Inf., vol. 33, no. 4, pp. 351–385, 1996.
[19] G. Graefe, "Partitioned b-trees - a user's guide," in BTW, 2003.
[20] ——, "Sorting and indexing with partitioned b-trees," in CIDR, 2003.
[21] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh, "Lazy-adaptive tree: An optimized index structure for flash devices," PVLDB, 2009.
[22] M. L. Kersten and S. Manegold, "Cracking the database store," in CIDR, 2005.
[23] S. Idreos, M. L. Kersten, and S. Manegold, "Self-organizing tuple reconstruction in column-stores," in SIGMOD, 2009.
[24] G. Graefe and K. Harumi, "Adaptive indexing for relational keys," in SMDB, 2010.
[25] M. Stonebraker, "The case for partial indexes," in VLDB, 1987.
[26] P. Seshadri and A. Swami, "Generalized partial indexes," in ICDE, 1995.
[27] S. Agrawal, S. Chaudhuri, L. Kollar, and V. Narasayya, "Index tuning wizard for microsoft sql server 2000," http://msdn2.microsoft.com/en-us/library/Aa902645(SQL.80).aspx.
[28] "http://www.postgresql.org/docs/current/static/ddl-partitioning.html," http://www.postgresql.org/docs/current/static/ddl-partitioning.html.
[29] H. Kimura, S. Madden, and S. B. Zdonik, "UPI: A Primary Index for Uncertain Databases," in VLDB, 2010.
[30] "Postgresql 8.1.20 documentation," http://www.postgresql.org/docs/8.1/static/release-8-1.html.
[31] D. Comer, "Ubiquitous B-Tree," in ACM Computing Surveys, vol. 11, no. 2, 1979.
[32] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. Zdonik, "CORADD: Correlation aware database designer for materialized views and indexes," in PVLDB, 2010.