Shinobi: Insert-aware Partitioning and Indexing Techniques For Skewed Database Workloads

by

Eugene Wu

B.S., University of California (2006)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2010

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2010

Certified by .....
Samuel Madden
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Shinobi: Insert-aware Partitioning and Indexing Techniques For Skewed Database Workloads

by

Eugene Wu

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2010, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Many data-intensive websites are characterized by a dataset that grows much faster than the rate that users access the data and possibly high insertion rates. In such systems, the growing size of the dataset leads to a larger overhead for maintaining and accessing indexes even while the query workload becomes increasingly skewed. Additionally, the database index update costs can be a non-trivial proportion of the overall system cost. Shinobi introduces a cost model that takes index update costs account, and proposes database design algorithms that optimally partition tables and drop indexes from partitions that are not queried often, and that maintain these partitions as workloads change. We show a 60× performance improvement over traditionally indexed tables using a real-world query workload derived from a traffic monitoring application and over 8× improvement for a Wikipedia workload.

Thesis Supervisor: Samuel Madden
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Load the turntables, it's about to get heavy
Got a grip of acknowledgments, so set your brain to READY

Sam Madden is the man, keeps my head on tight
Lets my mind fly high, and keeps the ground in sight
Dropping mad knowledge bombs, ain't an easy feat
Without his help and guidance, this thesis wouldn't be complete

Expounding for all these years, of education's might
My family worked hard, made sure I grew up right
Especially my mother, blows up my phone to say
Eat, drink and play, and sleep early everyday

G930 colleagues, their patience has been epic
Playing ball with ideas, while they put up with my antics

And to all my friends, who kept my life on track
Chatting, laughing, hanging, wouldn't take it back
To them I leave one last nugget – the world is just a snack
Eyes on the prize, baby

Eyes on the prize.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As many websites have scaled to tens or hundreds of thousands of users, data-intensive sites that include user-generated data or monitoring applications have found that their rate of data growth is out-pacing the growth in the rate at which users access data. For example, between 2003 and 2007, the number of english Wikipedia [1, 3] article revisions grew from 6× to 42× the number of articles, whereas users only access the most recent article revisions (Figure 1-1). Similarly, Cartel [24] is a sensor-based system we have built for collecting data from cars as they drive around Boston. The *centroidlocations* table stores timestamp, latitude, longitude, and car id information of the participating cars every second. The table has grown to over 18 GB in a few years, yet the majority of the queries only access data within the last day (Figure 1-2).

The workloads of such applications exhibit several interesting characteristics. First, the queries are both highly skewed and highly selective. In our Wikipedia snapshot, the *revision* table stores a record describing every article edit (excluding the article content), yet the application primarily accesses the records corresponding to the latest article revisions (5% of the table) using id look-ups. Similarly, the Cartel workload only accesses 5% of the *centroidlocations* table on any given day. About 80% of the queries go to the most recently inserted data (from the current day), and the average selectivity of the timestamp predicate is 0.1%. This characteristic suggests that only a small portion of the tables need to be

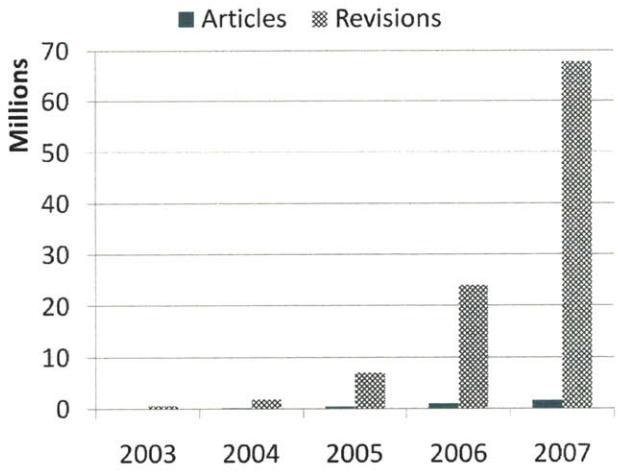


Figure 1-1: Number of Wikipedia articles and article revisions from 2003 to 2007

indexed to ensure high query performance.

Second, the data that the workload accesses is relatively predictable and grows slowly as compared to the rate of the insertions. In the case of Wikipedia, the set of queried revision data (one record for every article) increased 15×, whereas the total number of revisions grew by 107×. The difference is more substantial in the Cartel application, where the data grows daily yet the workload primarily targets the last day of data. This characteristic suggests that identifying and keeping track of the set accessed records that is orders of magnitude smaller than the total dataset size could be highly profitable.

Third, monitoring type applications may see insert rates that far exceed the query rate. Cartel's *centroidlocations* table serves approximately 200,000 inserts and 500 queries per day, or an insert-to-query ratio of 400:1. We expect that similar disparities exist in many web or stock tick monitoring applications, where data arrives continuously but is queried only a few times per day, with more queries going to more recent data or particular sites or symbols of interest. This characteristic indicates that inserts may be a nontrivial component of a workload.

A common way to make workloads run fast is to use database experts or index selection tools to analyze the query workload and install a recommended set of indexes. This approach

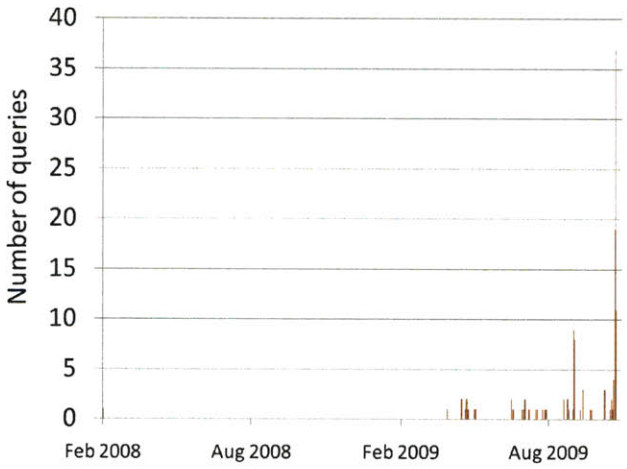


Figure 1-2: Frequency of queries in sub-ranges of Cartel's *centroidlocations* table on Nov. 19, 2009

trades off updating the indexes on every insert for improved query performance by reducing record access costs to a small number of index look-ups. For small datasets or workloads with a modest number of inserts, this dramatically improves the performance of selective queries. However, as the indexes grow alongside the dataset, both insert and query performance are adversely affected. For example, adding a single index on the *centroidlocations* table can slow the insert rate by up to 44× if the insertions are uniformly distributed throughout the table. Additionally, queries on Wikipedia's revision table perform 4× slower when the entire table is indexed as compared to only indexing the queried items.

In this thesis, we describe Shinobi, a system that uses existing and new techniques to improve the performance of skewed query workloads such as the Wikipedia and Cartel workloads described above, while optimizing for insert costs. Shinobi uses three key ideas: first, it partitions tables, such that regions of the table that are frequently queried together are stored together, and regions that are infrequently queried are stored separately. 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 uses a cost-based partitioner, which, given a set of indexes, a query workload and machine statistics such as RAM and the table size, finds the optimal table partitioning and the best set of indexes to create (from a user-supplied set) for each partition. As the workload evolves, Shinobi performs the minimum amount of repartitioning necessary to re-optimize the system for the new workload characteristics.

In summary, our contributions include:

1. *Cost-based selective indexing.* By taking insert costs into account, Shinobi chooses the optimal partitions to index and dramatically improves insert performance. In our experiments using a Cartel workload, Shinobi reduces index costs by 25× and query costs by more than 90× as compared to an unpartitioned, fully indexed, table.
2. *Cost-based partitioning.* Shinobi optimally partitions tables for a given workload in order to maximize query and insert performance.
3. *Adaptive re-partitioning* Shinobi dynamically adjusts the partitioning as the query workload changes, while minimizing the total expected cost.

Shinobi's advantages are most pronounced for monitoring workloads with relatively uniform inserts throughout the table and skewed, relatively selective queries. However, for other types of workloads, Shinobi will always out-perform an unpartitioned, fully-indexed, table.

Chapter 2

Related Work

There is a large body of work in the areas of stream processing, automated index selection and partitioning, index optimization, and partial indexes that both inspired, and are similar to, ideas in Shinobi.

2.1 Stream Processing Systems

Stream processing systems such as [4, 11, 9, 17, 35] are a special case of insert heavy workloads. These systems model the data source as a stream of records. Queries are compiled into query plans, which are probed by new records. A record that satisfies a query plan is output as a result. Much of the work has focused on reducing the time between a record entering the system and determining the set of queries it satisfies and on single-pass algorithms for computing various statistics over these streams. Although these systems are able to sustain high insert loads, they have limited support for querying historical data, typically relying on a traditional database system for historical queries. Many monitoring workloads, such as Cartel, need to be able to run these types of historical queries.

2.2 Database Designers

Many database designers use query optimizer extensions to perform *what if* analysis [14] – at a high level, the optimizer accepts hypothetical table configurations and queries as input and outputs the expected workload performance. The optimizer's wealth of statistics and its highly tuned cost model are powerful tools for accurately estimating a potential workload. Shinobi presents a cost model that does not attempt to replicate decades of optimizer research [26, 12], but rather identifies an essential set of parameters for evaluating range partitioning and selective indexing techniques on a mixed query and insert workload.

Index selection tools such as [7, 34, 13] introduce heuristics that explore the space of potential indexes and materialized views. Such tools find an optimal set of indexes within maximum size constraints. Shinobi analyzes the results of such tools (or hand-crafted physical designs), and uses index selection and partitioning techniques to identify subsets of a table where installing an index will be detrimental to performance. The end result is the ability to install more indexes within the same space constraints

Partitioning techniques such as [16, 10, 31, 8] 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.

2.3 B-tree Optimizations

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 [21, 27, 30], and Partitioned B-trees [19, 20], for traditional disk based systems, and flash optimized B-trees such as [5]. 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 large insert

workloads.

2.4 Adaptive Databases

Database Cracking [28, 25] and other adaptive indexing techniques incrementally sort and index the underlying table based on the query workload. Database cracking creates a copy of the keyed column and incrementally sorts the column using the results of workload queries. It 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. [22, 23] describe similar strategies for databases using B-trees and block-oriented (e.g., disk) storage. Fundamentally, these approaches fully index the table or column and would still benefit from dropping indexes from unqueried data ranges.

2.5 Partial Indexes

Finally, partial indexes [33] are a method for building an index on a predicate-defined subset of a table. Seshadri and Swami [32] 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 test. 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 6.2.6, 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 each query on average in 0.1-0.8 seconds. Thus, one way to view our work is as an index designer and efficient implementation of clustered,

non-overlapping partial indexes.

Chapter 3

Architecture

As noted above, Shinobi partitions and indexes tables to efficiently process workloads with a high insert-to-query ratio. Specifically, Shinobi finds the optimal set of non-overlapping range partitions and chooses indexes for each partition (together denoted as the *table configuration*) that maximize workload performance.

The input to Shinobi's designer 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., SQLServer Database Engine Tuning Adviser [6]). Queries are assumed to consist of range scans over the partitioning attributes while joins can be decomposed into a set of range queries for each of the tables participating in the join. For example, a GPS coordinate table for a city may partition the table on the lat, lon columns, which are used to answer spatial-temporal queries for certain locations and times, with new GPS points being inserted throughout the city.

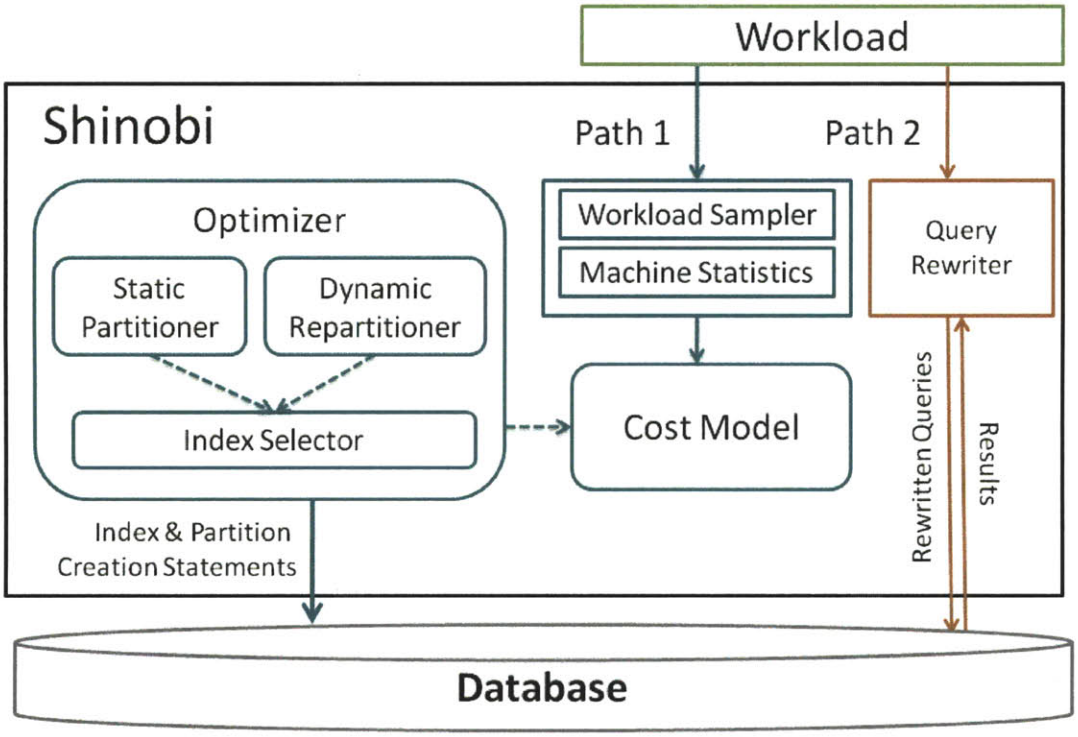


Figure 3-1: The Shinobi architecture

3.1 Overview

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 3-1 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 rewrites queries to execute on the partitioned table, and redirects

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 (w/out index updates)
$icost_{overhead}$	.019ms/MB	insert overhead per MB of indexes
$lifetime_W$	variable	Expected # queries in workload W

Table 3.1: Workload statistics and corresponding values in Cartel experiments

inserts to their relevant partitions.

The Workload Sampler reads recent SQL statements from the query stream and computes workload characteristics such as the query to insert 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 3 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, Static Partitioner, and Dynamic Repartitioner components all use the Cost Model to optimize the table configuration. Index Selection calculates the best set of indexes to install on each partition of a table. The Static Partitioner finds an optimal partitioning for the table and calls the Index Selector to build indexes. Finally, the Dynamic Repartitioner re-optimizes the table configuration as the workload varies.

Chapter 4

Cost Model

In this chapter, we introduce an insert-aware cost model and a model to predict repartitioning costs for a single table. The first estimates the average cost for a statement in a single table workload while the second estimates the cost to switch between different table configurations. This cost model is used in Section 5 to choose the optimal partitioning and index configuration.

We present the necessary components to accurately predict the relative differences between table configurations, and refer the reader to decades of query optimization research [26, 12] for further refinements of the model. For workload analysis purposes, we treat joins as a set of range queries over the tables involved in the join (fundamentally, there is no restriction on the types of queries the system can support).

4.1 Variables

The values of the model constants were derived experimentally and are shown in Table 3. Additionally, the following is a list of common variables used throughout the rest of this thesis. 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.

4.2 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,

$$cost(\Pi, \Psi) = |W_q|/|W| * cost_{select}(\Pi, \Psi) + |W_i|/|W| * cost_{insert}(\Psi) \tag{4.1}$$

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

4.2.1 Select Costs

The main components that determine select cost are the cost of index and sequential scans over each partition. We make the simplifying assumptions that all indexes are unclustered, a query q uses an index in ψ_P if it exists, and the cost is proportional to the amount of

data being accessed. 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:

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

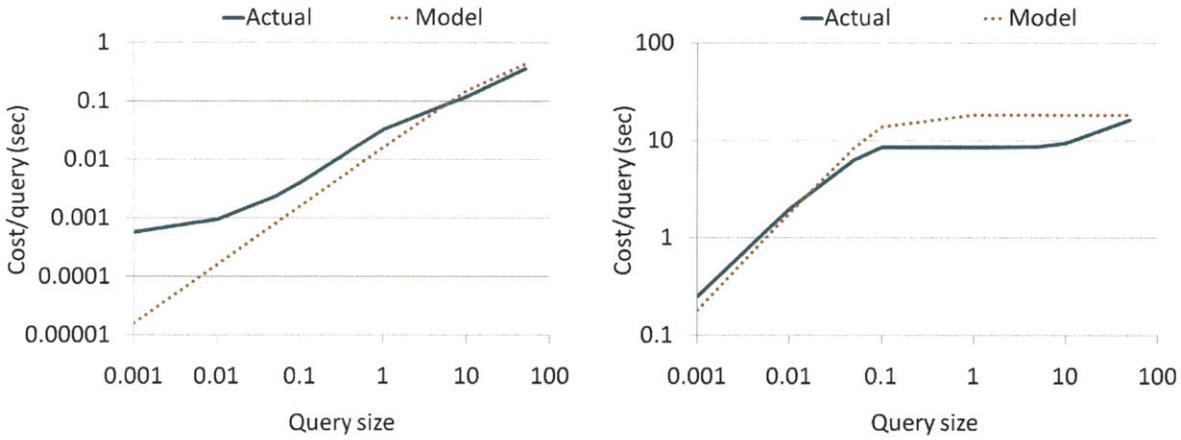
$cost_{pselect}$ is the average cost per query in W_{qp}. *seqscan* is the cost of a sequential scan, and modeled as the sum of the seek cost plus the cost of reading the partition:

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

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

indexscan 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. We assume that the database system is using bitmap scans that sort the page ids before accessing the heap file [2]. 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:

$$indexscan(s,p) = seqscan(p) \times \frac{1 - e^{-k \times s}}{1 + e^{-k \times s}}$$



(a) size(table) < RAM

(b) size(table) >= RAM

Figure 4-1: Query cost w.r.t. query selectivity

Figure 4-1 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 4-1(a), the model under-estimates the cost for very small queries and over-estimates the cost for queries larger than .1% in Figure 4-1(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.

4.2.2 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. We first describe how to model the cost of inserting into a single

partition, followed by a model for multiple partitions.

Single Partition

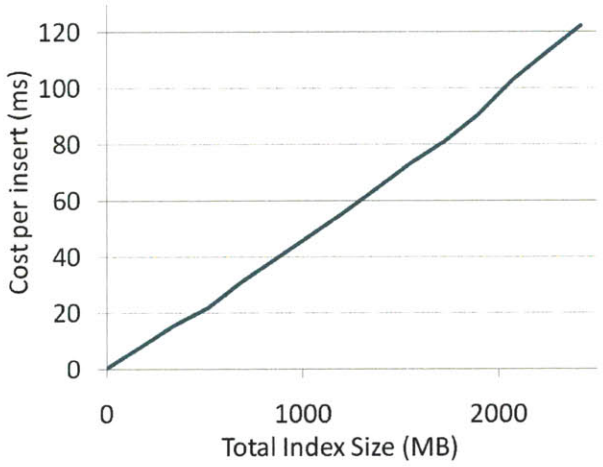


Figure 4-2: Insert cost grows linearly w.r.t. index size

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 (Figure 4-2). The overhead is the product of the cost of updating a one MB index, $icost_{overhead}$, and the total size of all indexes on the partition:

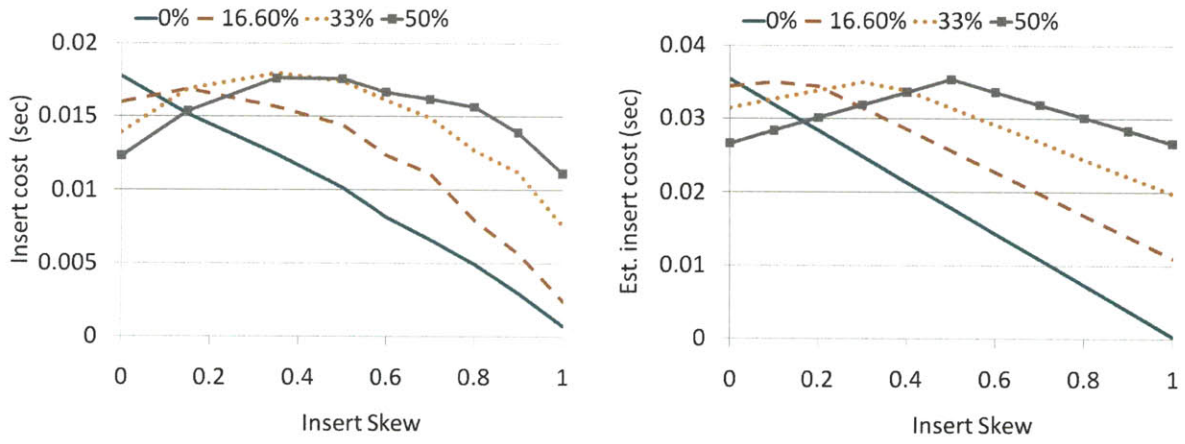
$$cost_{insert}(\psi_i) = icost_{fixed} + icost_{overhead} \times \sum_{uin\psi_i} size(u)$$

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 [15]. Figure 4-2 shows that in 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. We observed similar behavior in [29].

Two Partitions



(a) Actual costs

(b) Model estimates

Figure 4-3: Insert cost w.r.t. data (curves) and insert skew (x-axis)

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; 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 is dependent 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 4-3 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.

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:

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

and the complete insert cost model is:

$$cost_{insert}(\Psi) = icost_{fixed} +$$
$$icost_{overhead} \times \left(\sum_{\psi_k \in \Psi} size(\psi_k) - \sum_{\psi_j \in \Psi} max\left(0, \left(size(\psi_j) - \sum_{\psi_k \in \Psi} size(\psi_k) * \frac{|W_i \cap \psi_j|}{|W_i|} \right)\right)\right)$$

4.2.3 Limitations

It is important to note that the goal of our cost model is not to precisely estimate the expected cost of range queries – there are far too many parameters (e.g., clustered indexes, caching effects) and nuances in the query executor for us to completely model. Our goal is to accurately order the query performance of different table configurations, and as our experiments validate, the simplified cost model is enough to achieve this goal and allow us to see large performance gains.

4.3 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 experimentally 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.

4.3.1 Partition Costs

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

$$repart_{part}(\Pi_{old}, \Psi_{old}, \Pi_{new}) = \sum_{p \in \Pi_{new}} createp(p, \{(p_i, \psi_i) \in (\Pi_{old}, \Psi_{old}) | p_i \cap p \neq \emptyset \wedge p_i \neq p\})$$

$$createp(p, \Lambda_{\cap}) = \sum_{p_{\cap}, \psi_{\cap} \in \Lambda_{\cap}} (cost_{pselect}(W_{create,p\cap}, p_{\cap}, \psi_{\cap}) + \frac{size(|p_{\cap} \cap p|)}{cost_{dbcopy}})$$

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.

4.3.2 Indexing Costs

The cost of installing indexes is directly proportional to the size of the partition being indexed:

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

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:

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

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

4.3.3 Total Cost

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

$$repart(\Pi_{old}, \Psi_{old}, \Pi_{new}, \Psi_{new}) = repart_{part}(\Pi_{old}, \Psi_{old}, \Pi_{new}) + repart_{idx}(\Pi_{old}, \Psi_{old}, \Pi_{new}, \Psi_{new})$$

4.4 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.

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

$lifetime_W$ is the expected lifetime, in number of queries, of the current 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, $lifetime_{Wq}$, and the insert only workload, $lifetime_{Wi}$.

$$lifetime_W = lifetime_{Wq} + lifetime_{Wq}$$

In Section 5.4, we present an online algorithm that learns the expected lifetime of a query-only or insert-only workload and test its effectiveness in Section 6.2.5.

4.4.1 Total Workload Benefit

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

$$benefit_{total}(\Pi_{old}, \Psi_{old}, \Pi_{new}, \Psi_{new}) = benefit_{W}(\Pi_{old}, \Psi_{old}, \Pi_{new}, \Psi_{new}) - repart(\Pi_{old}, \Psi_{old}, \Pi_{new}, \Psi_{new})$$

Chapter 5

Optimizers

This chapter describes Shinobi's three primary optimizers that use the cost model, a strategy for estimating the value of $lifetime_W$, and a method called *record shuffling* that physically clusters predictably queried data. The optimizers partition the table, select indexes for each partition, and repartition the table when the workload changes. They use greedy algorithms designed to be simple to implement while still producing good results. We first describe the *Index Selector*, because it is used by both the *Static Partitioner* and the *Dynamic Repartitioner*.

5.1 Index Selector

The goal of *Index Selector* is to find the Ψ that minimizes the expected cost of a statement given W and Π. Formally, the optimization goal is:

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

Finding the naive solution to this optimization problem requires an exhaustive search ($O(2^{|\Pi|*|I|})$). This is because the indexes do not independently affect the cost model – since the number of inserts and queries to each partition are different, every possible combination of indexes needs to be considered. We instead consider a greedy approach that adds a small number

of indexes at a time, stopping once a local maximum is reached. [13] uses this algorithm (called Configuration Enumeration) to pick k indexes from a large set of possible indexes that will most improve a given workload.

1: $\Psi \leftarrow \{\{\}|0 \leq i \leq |\Pi|\}$
2: allpairs $\leftarrow \{(I_j, \psi_j)|I_j \in I \wedge \psi_j \in \Psi\}$
3: $\text{cost}_{\text{best}} \leftarrow \text{cost}(\Pi, \Psi)$
4: **while** true **do**
5: $\quad \Psi_{\text{best}}, \text{V}_{\text{best}} \leftarrow \text{null}, \text{null}$
6: $\quad$ **for** $\text{V} \in \{\text{v} \subseteq \text{allpairs}|0 < |\text{v}| \leq \text{k}\}$ **do**
7: $\quad\quad \Psi' \leftarrow \{\psi_1, .., \psi_\text{N}\}$
8: $\quad\quad$ **for** $(\text{I}_\text{j}, \psi_\text{j}) \in \text{V}$ **do**
9: $\quad\quad\quad \Psi' \leftarrow \{\psi_1, .., \psi_\text{j} \cup \{\text{I}_\text{j}\}, ..\}$
10: $\quad\quad$ **end for**
11: $\quad\quad$ cost' $\leftarrow \text{cost}(\Pi, \Psi')$
12: $\quad\quad$ **if** $\text{cost}' < \text{cost}_{\text{best}}$ **then**
13: $\quad\quad\quad \Psi_{\text{best}}, \text{V}_{\text{best}}, \text{cost}_{\text{best}} \leftarrow \Psi', \text{V}, \text{cost}'$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: $\quad$ **if** $\text{V}_{\text{best}} = \text{null}$ **then**
17: $\quad\quad$ **return** Ψ
18: $\quad$ **end if**
19: $\quad \Psi \leftarrow \Psi_{best}$
20: $\quad$ allpairs $\leftarrow$ allpairs $\setminus \text{V}_{\text{best}}$
21: **end while**

Algorithm 1: SelectIndex(Π,k)

SelectIndex (Algorithm 1) takes as input a partitioning Π and a look-ahead value k and iteratively adds the most beneficial k indexes until additional indexes does not improve the performance. Ψ keeps track of the installed indexes; initially no indexes are selected. *allpairs* contains all pairs of indexes and partitions where the index can be installed on the partition, and $cost_{best}$ keeps track of the best index configuration so far. In each iteration (line 4), we pick up to the best k indexes to install. To do this, we consider every combination of k or fewer indexes (line 6) and compute the expected cost, $cost'$, after installing them (lines 8-11). We keep track of the best indexes to install if $cost'$ is an improvement over the best configuration so far (line 12,13). If none of the indexes improve the cost, the algorithm exits (line 17), otherwise Ψ is updated with the best indexes in this iteration (line 19) and the

installed indexes are removed from future consideration (line 20).

The look-ahead value, k, is a parameter that dictates how thoroughly to explore the search space. For a given k, the algorithm runs for up to $(|\Pi||I|)/k$ iterations, and each iteration examines all subsets of *allpairs* up to size k. Thus, *SelectIndex* has a runtime of $O(\frac{|\Pi||I|}{k} \times \sum_{n=0}^{k} \binom{|\Pi||I|}{k})$. 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 $O((|\Pi||I|)^2)$.

We use two additional optimizations to improve the speed of this algorithm. First, we don't install indexes on any partitions that are not queried. Second, when we find an index-partition pair that does not improve the expected cost, we remove it from any further consideration. We can do this because as additional indexes are built, the cost of maintaining the indexes strictly increases while the query benefits stay constant.

5.2 Static Partitioner

Shinobi's partitioning component searches through a space of possible partitions to find an optimal partitioning for a workload W. The goal of the partitioning is two-fold. First, it separates the queried and unqueried ranges of data, which enables Shinobi to drop indexes on infrequently queried data. Second, partitioning increases query performance by reducing sequential scan costs and improving locality for queries that use indexes. Our experiments in Section 6.2.4 show that we meet these two goals and can find the optimal table configuration automatically.

StaticPartition (Algorithm 2) uses an N-dimensional quad-tree and splits the leaf nodes if the query performance is expected to improve. A quad-tree [18] is a tree data structure where internal nodes have four children and is used for partitioning a finite 2D space – the tree recursively sub-divides the space into four quadrants. An N-dimensional quad-tree recursively subdivides space into 2^N non-overlapping regions and each internal node has up to 2^N children. The output of the *StaticPartition* algorithm is a partitioning that the optimizer uses to generate SQL statements that actually partition the data. The dimensionality of the tree is equal to the number of partitioning attribute(s). Each leaf node represents a

single partition containing a sub-range of the data. The tree also implements the method *getPartitions*(), which returns the partitioning represented by the leaf nodes.

Note that in the following pseudocode "split" simply refers to choosing a node to split in the in-memory quad tree (which requires just a few CPU instructions), not actually splitting the data on disk in the database, which is very slow and is only done after the optimal configuration is found.

Starting with a single node tree, we populate the queue *nodes* with the root node and generate the associated partitioning and indexes, Π and Ψ (lines 1-4). In each iteration, the algorithm removes a node from *nodes* (line 6) and calculates the table configuration if the node were split (lines 7-9). If the split will reduce the workload cost, we split the leaf, append its children to *nodes*, and update the current partitioning and indexes (lines 10-12), otherwise we restore the tree (line 14). The algorithm continues until further splitting degrades the expected performance, after which it returns the final partitioning.

```
1: root ← create tree
2: nodes ← {root}
3: Π ← root.getPartitions()
4: Ψ ← SelectIndex(Π,1)
5: while |nodes| > 0 do
6:    leaf ← nodes.pop()
7:    leaf.split()
8:    Π′ ← root.getPartitions()
9:    Ψ′ ← SelectIndex(Π′,1)
10:   if benefit_w(Π, Ψ, Π′, Ψ′) > 0 then
11:      Π, Ψ ← Π′, Ψ′
12:      nodes.pushall(leaf.children)
13:   else
14:      leaf.merge()
15:   end if
16: end while
17: return root.getPartitions()
```

Algorithm 2: StaticPartition()

5.3 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 4.4) to evaluate whether a new configuration is worth the repartitioning cost.

This component uses a tree based approach similar to *StaticPartition* in the previous section. The main differences are that its optimization function takes into account repartitioning costs and that it takes as input the tree representation of the current partitioning, *root*, and the current indexing, Ψ. 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 component then calculates the optimal indexing by calling *SelectIndex*. The order of the phases is not important [1].

The goal of the merging phase (Algorithm 3) is to find the set of nodes to merge that will maximize the expected benefit (as defined in 4.4) 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 set of intermediate nodes that are the parents of the leaves (line 4). The algorithm searches for the node to merge that will maximize the benefit over $benefit_{best}$ (lines 6-15). This is done by temporarily merging 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) by re-splitting the node. 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 and $benefit_{best}$ is updated to the benefit of the new partitioning (lines 19-20). The runtime of this algorithm is limited by the number

[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.

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 splitting phase (Algorithm 4) is nearly identical to the merging phase except the goal is to find the set of splits (rather than merges) that will maximize the expected benefit. In every iteration, the algorithm checks all of the leaves to find the node that will maximize the expected benefit (section 4.4) when split. The algorithms only differ in minor instances (highlighted in italics). First, *nodes* is defined as the leaves rather than the parents of the leaves (line 4). Second, the nodes are split instead of merged and vice versa (lines 7,11,19).

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:  Π ← root.getPartitions()
2:  benefit_best ← 0
3:  while true do
4:    nodes ← {l.parent|l ∈ root.leaves()}
5:    benefit, node ← 0, null
6:    for n ∈ nodes do
7:      n.merge()
8:      Π′ ← root.getPartitions()
9:      Ψ′ ← SelectIndex(Π′,1)
10:     benefit′ = benefit(Π, Ψ, Π′, Ψ′)
11:     n.split()
12:     if benefit′ > benefit ∧ benefit′ > benefit_best then
13:       benefit, node ← benefit′, n
14:     end if
15:   end for
16:   if node = null then
17:     return root
18:   end if
19:   node.merge()
20:   benefit_best ← benefit
21: end while
```

Algorithm 3: MergePartitions(root, Ψ)

```
1: Π ← root.getPartitions()
2: benefit_best ← 0
3: while true do
4:   nodes ← root.leaves()
5:   benefit, node ← 0, null
6:   for n ∈ nodes do
7:     n.split()
8:     Π' ← root.getPartitions()
9:     Ψ' ← SelectIndex(Π',1)
10:    benefit' = benefit(Π, Ψ, Π', Ψ')
11:    n.merge()
12:    if benefit' > benefit ∧ benefit' > benefit_best then
13:      benefit, node ← benefit', n
14:    end if
15:  end for
16:  if node = null then
17:    return root
18:  end if
19:  node.split()
20:  benefit_best ← benefit
21: end while
```

Algorithm 4: SplitPartitions(root, Ψ)

5.4 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 main 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 moving sum of the individual lifetimes in the vector. The lifetime of a

partition is calculated as the average lifetime of the intersecting ranges.

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. In order to avoid penalizing the case where r_i is not queried for many timesteps, v_i only appends to the vector 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, $\alpha \in [0, 1]$ and then incremented by the number of queries that access r_i in the current time interval. α controls the number of timesteps into the future for which a query is counted and is set to 0.2 in our experiments. 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. In our experiments we use $\tau = 0.01$, which we experimentally determined to work well.

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 set β to 0.2.

5.5 Record Shuffling

We noticed that many workloads are highly amenable to partitioning optimizations, but the physical distribution of the records sometimes limits the effectiveness of such strategies. In the Wikipedia workload (see Section 6.4 for details), 99.9% of the queries access only 5% of records in the revision table (those that correspond to the latest revision of an article).

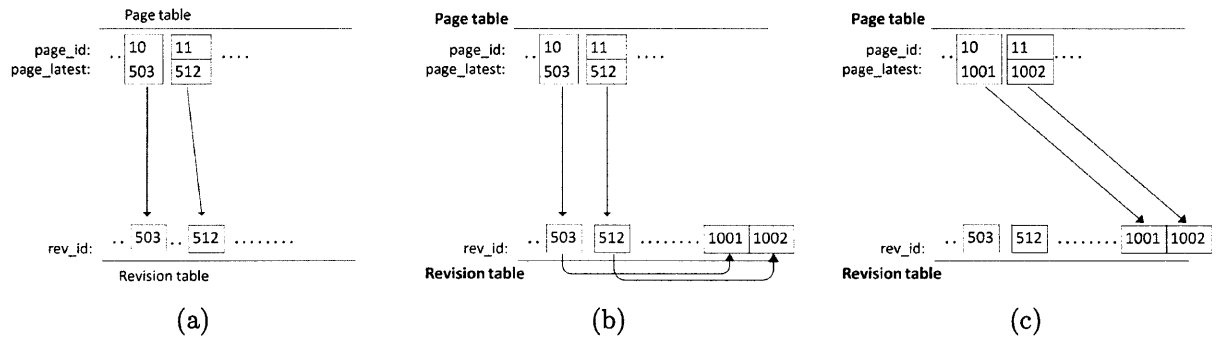


Figure 5-1: Record shuffling steps

Theoretically, Shinobi can partition the table into queried (hot) and non-queried (cold) partitions and see substantial performance gains, however, the accessed records are distributed throughout the table, making it very difficult to improve performance by partitioning the table into contiguous ranges. In this section, we introduce a *record shuffling* algorithm that exploits the auto-increment attribute found in many tables and reorganizes the records to enable effective partitioning. We first describe a single pass algorithm, then extend it to perform incremental shuffling, and finally describe maintenance strategies. We assume that the hot records can be or have been identified. For example, in the Wikipedia database, each article record contains a foreign key reference to the latest revision record. In Section 6.4, we show that record shuffling enables Shinobi to optimize one of Wikipedia's main tables and dramatically improves performance by over 8×.

The high level idea is to atomically copy the heavily accessed records to a different region so that they are clustered together, and update the foreign keys from other tables to refer to these new records. The different region may be a separate partition or the end of the table. This approach depends on the existance of a semantically meaningless field that identifies the records[2]; applications such as Wikipedia exhibit this property. Figure 5-1 depicts the process of shuffling records in a revision table to the end of the table and updating the foreign keys in a page table. In Figure 5-1(a), page records 10 and 11 respectively point to revision

[2]By semantically meaningless, we mean that the field is not used outside of the database other than as an opaque identifier. For applications that cache the field's value, it is possible to temporarily map the old field values to the new records until the caches are invalidated.

records 503 and 506 in the historical portion of the revision table. We shuffle the records by inserting copies of the revision records (figure 5-1(b)), and updating the pointers in the page table (figure 5-1(c)).

In many cases, shuffling the entire table at once will cause unacceptable performance penalties for the application (it took over 16 hours to shuffle the revision table). Fortunately this process can be performed over a long period of time by shuffling small ranges of the table at a time. It can run when the system has extra resources and be paused when the system is under load. Additionally, the system does not need to be quiesced because shuffling is implemented using normal SQL transactions.

As the application runs, new records will be inserted and hot records will become cold. When the ratio of hot to cold records in a range drops below a threshold, we can reshuffle the range and compact the hot records. Because we use a threshold, we can restrict the maximum number of record that will be moved for a given range. Finally, a periodic process scans the table for cold copies and deletes them.

Chapter 6

Experiments

In this section, we run three sets of experiments that demonstrate the utility of using Shinobi to partition and index tables and the resulting performance gains. The first two experiments use a modified Cartel data and workload. We analyze the applicability and effectiveness of Shinobi's cost model and online adaptive algorithms, and show that Shinobi reduces the time to process the workloads by an order of magnitude. The third set of experiments use a snapshot of Wikipedia's database and real-world query logs. The workload performs very few inserts, yet we show that Shinobi can use a record shuffling technique to take advantage of the query skew and dramatically reduce the time it takes to execute a fixed set of query traces.

The chapter is laid out as follows. Section 6.2 runs experiments using a one dimensional dataset and workload. We first show the best possible performance gains on a static workload in Section 6.2.4, then shift focus to optimizing dynamically changing workloads. Section 6.2.5 highlights the importance and accuracy of the $lifetime_W$ estimator, Section 6.2.6 validates the accuracy of the cost model, and Section 6.2.7 shows how Shinobi responds to workloads with different characteristics. We show that our optimizations scale to multiple dimensions in Section 6.3, and illustrate the effectiveness of record shuffling on a Wikipedia dataset in Section 6.4. Finally, we analyze the runtime of Shinobi's algorithms in Section 6.5.

6.1 Implementation

The current prototype is written in Python and issues SQL commands to a backend database (this work uses PostgreSQL and MySQL). Each partition is implemented as a separate table, and queries are rewritten to execute on the partitions.

The experiments in Sections 6.2 and 6.3 are run on a dual 3.2 GHz Pentium IV with 512 MB of RAM and a 300GB 7200 RPM drive, running Redhat Linux 2.6.16 and PostgreSQL. The experiments in Section 6.4 are run on a 16 core 2.27 GHz Intel Xeon with 24 GB of RAM and 6x2 TB 7200 RPM drives configured in hardware RAID5, running Ubuntu 9.10 and MySQL.

6.2 One-Dimensional Performance

In this section, we first show how Shinobi can use adaptive partitioning and selective indexing to realize more than an order of magnitude improvement on a workload consisting of skewed select queries and uniform inserts. Then, we evaluate how Shinobi's model based optimizations respond as different workload characteristics vary.

After describing the experimental dataset and workload, we first show how Shinobi optimizes a static workload. Then, we run Shinobi on adaptively changing workloads to illustrate the adaptive lifetime estimator and show that we reduce the cumulative cost of a Vehicular workload by an order of magnitude. Finally, we show how Shinobi decides to install or drop indexes as various workload and partitioning characteristics change.

6.2.1 Dataset

Each experiment is executed on a subset of the *centroidlocations* table from the Cartel database. Each record consists of lat, lon, timestamp, and several other identification attributes and the values of the timestamp attribute are uniformly distributed throughout the table. The table size is 3.4 GB, contains 30 million records, and is partitioned and indexed on the *timestamp* column.

6.2.2 Workloads

In the following experiments, we use a set of realistic Cartel workloads, $W_{cartel1D}$, and a set of synthetic workloads, $W_{synth1D}$. Both workloads consist of 10 timesteps worth of queries and inserts; each timestep contains a number of range queries followed by a large number of insertions uniformly distributed across the table. The queries access 0.1% of the table and have the form:

select * from centroidlocations where $[min]$ <= timestamp and timestamp < $[max]$;

The Cartel workload ($W_{cartel1D}$) is an example of a realistic query workload. The queries are 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 pick 360 inserts for each query (36k total). The queries in each timestep read the most recent 30% of data, but 9% periodically access random historical data.

The synthetic workload ($W_{synth1D}$) is an example workload to showcase how Shinobi responds to skewed workloads that shift the "hot spot" at varying intervals. Each timestep generates queries from a zipfian distribution (λ=1.5) centered around a random timestamp value and uniquely accesses 8% of the table. $W_{synth1D}$ has the same number of queries in each timestep, however the number of queries vary between 1 to 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.

6.2.3 Approaches

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 5.1 to only create beneficial indexes.

Static Partitioning (SP_X) partitions the table into X equally sized partitions, and Optimized Partitioning (OP) finds the optimal partitioning as described in Section 5.2.

We compare, fully indexed table ($FISI_1$); full and selective indexing on a table statically partitioned into 10 ($FISP_{10}, SISP_{10}$) and 50 ($FISP_{50}, SISP_{50}$) partitions; and selective indexing on an optimally partitioned table (*Shinobi*).

6.2.4 Static Optimizations

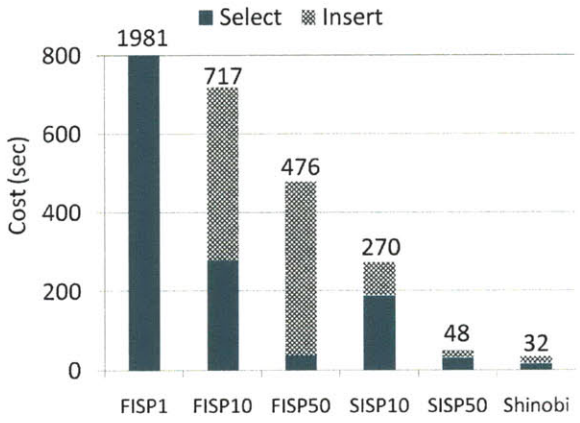


Figure 6-1: Select and insert costs for statically optimized $W_{cartel1D}$ workload

This experiment illustrates the benefits of partitioning and selectively indexing a table running workload $W_{cartel1D}$. Specifically, we are interested in achieving the best possible query and insert performance for a given workload, assuming the workload is provided ahead of time and that we can optimize the table prior to running the workload. In each approach, the optimizers analyze the workload and find the optimal table configuration. The reported results are the average cost of running all timesteps; the variance between timesteps was not significant.

Figure 6-1 shows that simply indexing an unpartitioned table results in suboptimal query and insert performance, due to the large number of seeks for each query and the large index size. Splitting the table into 10 ($FISP_{10}$) and 50 ($FISP_{50}$) partitions reduces select costs

by 5.5 and 42×, respectively. The query improvement is largely because each partition physically groups together data within the same logical range, so that an arbitrary range scan needs to read less pages of data. Selective indexing ($SISP_{10}$ and $SISP_{50}$) decreases insert costs by 4 and 20×. Shinobi finds the optimal table configuration that reduces select and insert cost by 97 and 25×, respectively, and uses the cost model to avoid a manual search process for the optimal partitioning. Overall Shinobi is about 60× faster than $FISP_1$.

Figure 6-2: Shinobi partitions and indexes for Nov. 19th workload

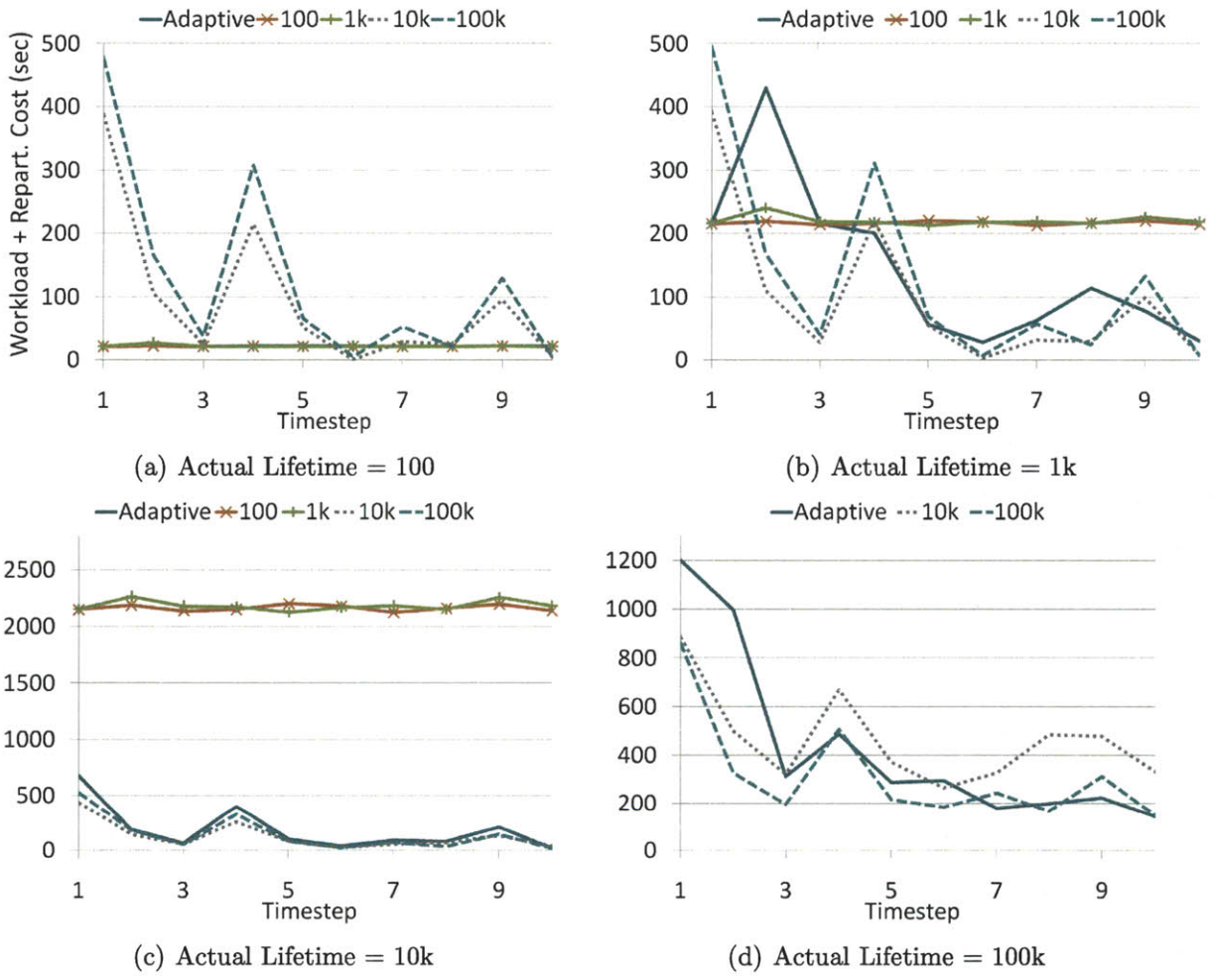
Figure 6-2 illustrates the table configuration produced by Shinobi's static partitioner for the Nov. 19 Cartel workload. The solid bars above the x-axis indicate the number of queries that access each time range of data in the *centroidlocations* table on Nov. 19, (where the leftmost portion is older historical data and the rightmost portion is data inserted on Nov. 19). The dashed bars separate adjacent partitions, the solid bar under the x-axis indicates that the partition above it is indexed.

We can see that the queried regions are heavily partitioned; small partitions result in faster sequential scans and a smaller need to index the partitions. Interestingly, the rightmost partition is accessed the most but is less partitioned and indexed. This is because the partition is dominated by query performance, so the query benefit of an index outweighs the insertion overhead. Conversely, the other regions are sparsely accessed and partitioning is a mechanism to avoid indexing such regions. It appears that queries will always access fresh data, so indexing the most recent data is the most effective strategy. However, we claim this is not generally applicable. For example, if the data is spatially partitioned, (Figure 6.3.4), *recent data* is not a meaningful concept whereas Shinobi can find an optimal partitioning

and indexing strategy.

We found $FISP_1$'s performance to be consistently poor in the other experiments and thus do not further include it in any other figures. Where appropriate, we provide a textual description of its performance.

6.2.5 Lifetime Estimation



(a) Actual Lifetime = 100

(b) Actual Lifetime = 1k

(c) Actual Lifetime = 10k

(d) Actual Lifetime = 100k

Figure 6-3: Shinobi performance with static and adaptive $lifetime_W$ values (curves) for different actual lifetimes (plots)

In this set of experiments, we analyze Shinobi's adaptivity to workloads ($W_{synth1D}$) of varying dynamism and show the importance of accurately predicting the value of $lifetime_W$.

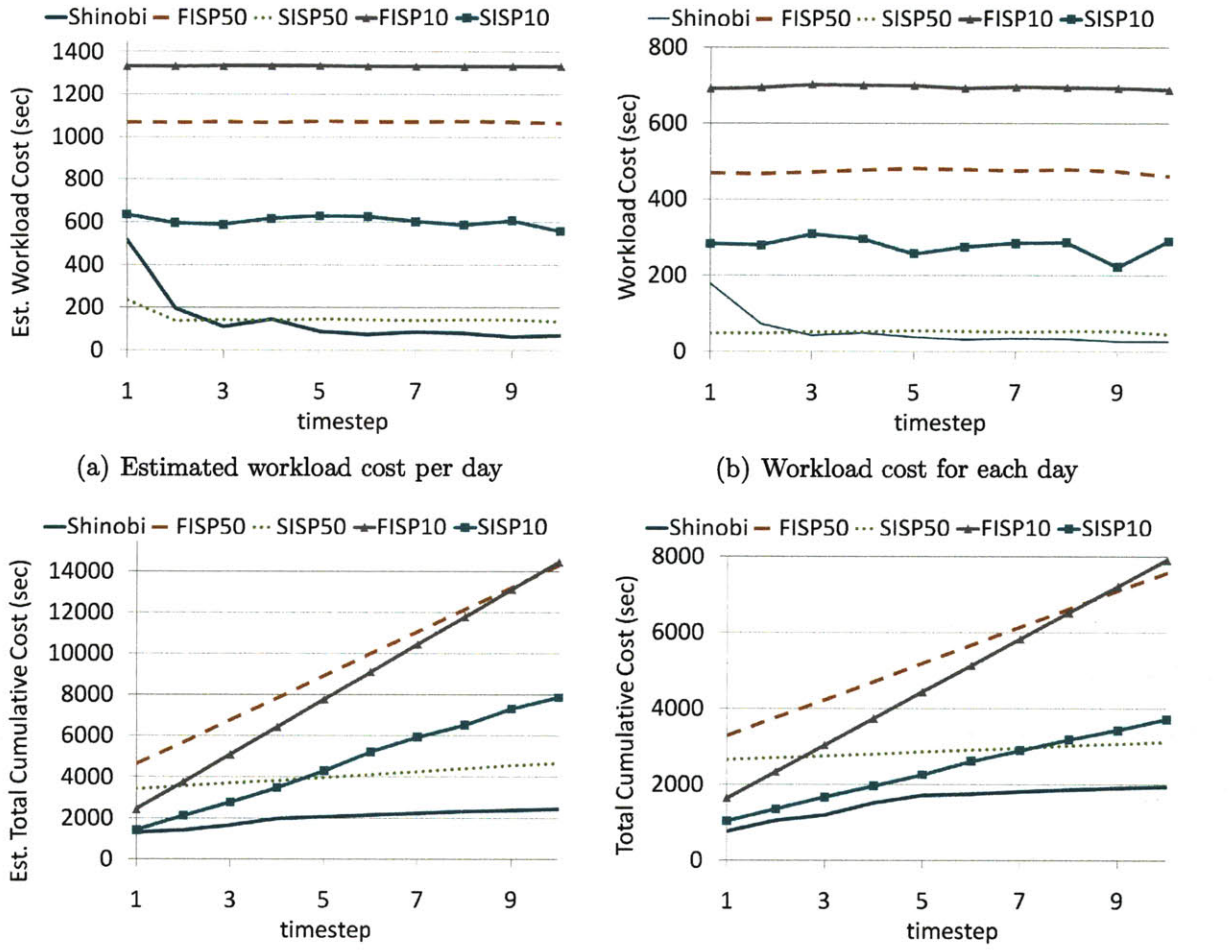
We show that Shinobi running with the adaptive $lifetime_W$ estimator performs comparably to a "lookahead" Shinobi that is able to analyze the queries in a timestep to determine the lifetime prior to executing it (while the real algorithm must adaptively estimate it). The lookahead Shinobi is configured with static $lifetime_W$ values ranging from 100 to 100k. Each experiment executes one workload per timestep for 10 timesteps.

In each timestep, the lookahead-based approaches load the new workload, run the repartitioning algorithm using the given $lifetime_W$ value, and execute the workload to completion. The adaptive approach estimates the new $lifetime_W$ in an online fashion (see Section 5.4 for algorithm details).

Figure 6-3 shows the workload plus repartitioning costs at each timestep when the workload lifetime is 100, 1k, 10k and 100k SQL statements. We first discuss the lookahead curves and then examine the performance of the adaptive algorithm. 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 6-3(b), where the 10k curve outperforms the 1k curve. The reason is that the naive approach disregards the fact that two consecutive workloads are likely to overlap, and therefore may underestimate $lifetime_W$ for shorter workloads. In general we found that it is better to over-estimate $lifetime_W$ and over-partition the table, rather than to run every query sub-optimally.

The adaptive $lifetime_W$ estimator (Adaptive) performs competitively in all of the experiments. In Figure 6-3(a), its curve is identical to the 10 and 1k curves and in the other experiments, it converges to the optimal curve within 4 timesteps. 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. During this period, the query performance can be suboptimal and Shinobi may repartition the same set of data several times. These two factors contribute to the spikes seen in the Adaptive curves, such as in timestep 2 of Figures 6-3(b) and 6-3(d), where the cost is quite high due to repartitioning costs and sub-optimal queries.

6.2.6 Cost Model Validation



(a) Estimated workload cost per day

(b) Workload cost for each day

(c) Estimated total cumulative cost

(d) Actual total cumulative cost

Figure 6-4: Performance over 10 timesteps of the Cartel workload

In this experiment we run Shinobi on a realistic workload ($W_{cartel1D}$) 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 6-4(b) shows the workload only performance over the 10 days; the performance is similar to the results in section 6.2.4. For reference, $FISP_1$ took on average 2000 sec per timestep. The $FISP_{10,50}$ curves illustrate the effectiveness of statically partitioning the table

into 10, and 50 partitions, respectively. Increasing the number of partitions from 1 to 10 is enormously beneficial – reducing the average cost by nearly 3× – while increasing to 50 partitions reduces the cost by 4×. Selective indexing only creates indexes on heavily queried partitions, and reduces insert costs for $SISP_{10,50}$ by a factor of 10. In fact, for days 5-7, $SISP_{50}$ didn't create any indexes. Shinobi performs as well as $SISP_{50}$; 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 optimizes the table so that Shinobi performs very close to $SISP_{50}$ by day 2 and slightly out-performs $SISP_{50}$ by day 3.

Figure 6-4 plots the estimated and real cumulative cost of partitioning the table and running the workloads. For reference, $FISP_1$ took 20,000s to run the experiment. The values on timestep=1 are dominated by the initial partitioning and indexing costs. Splitting the table into 10 and 50 partitions costs 660 and 2500s, respectively, while indexing all of the data costs 240s. Although selective indexing ($SISP_{10,50}$) can avoid indexing a large fraction of the partitions and reduce indexing costs by almost 200s, these initial partitioning costs are still substantial. In contrast, Shinobi chooses a cheap partitioning because the estimated $lifetime_W$ is still low.

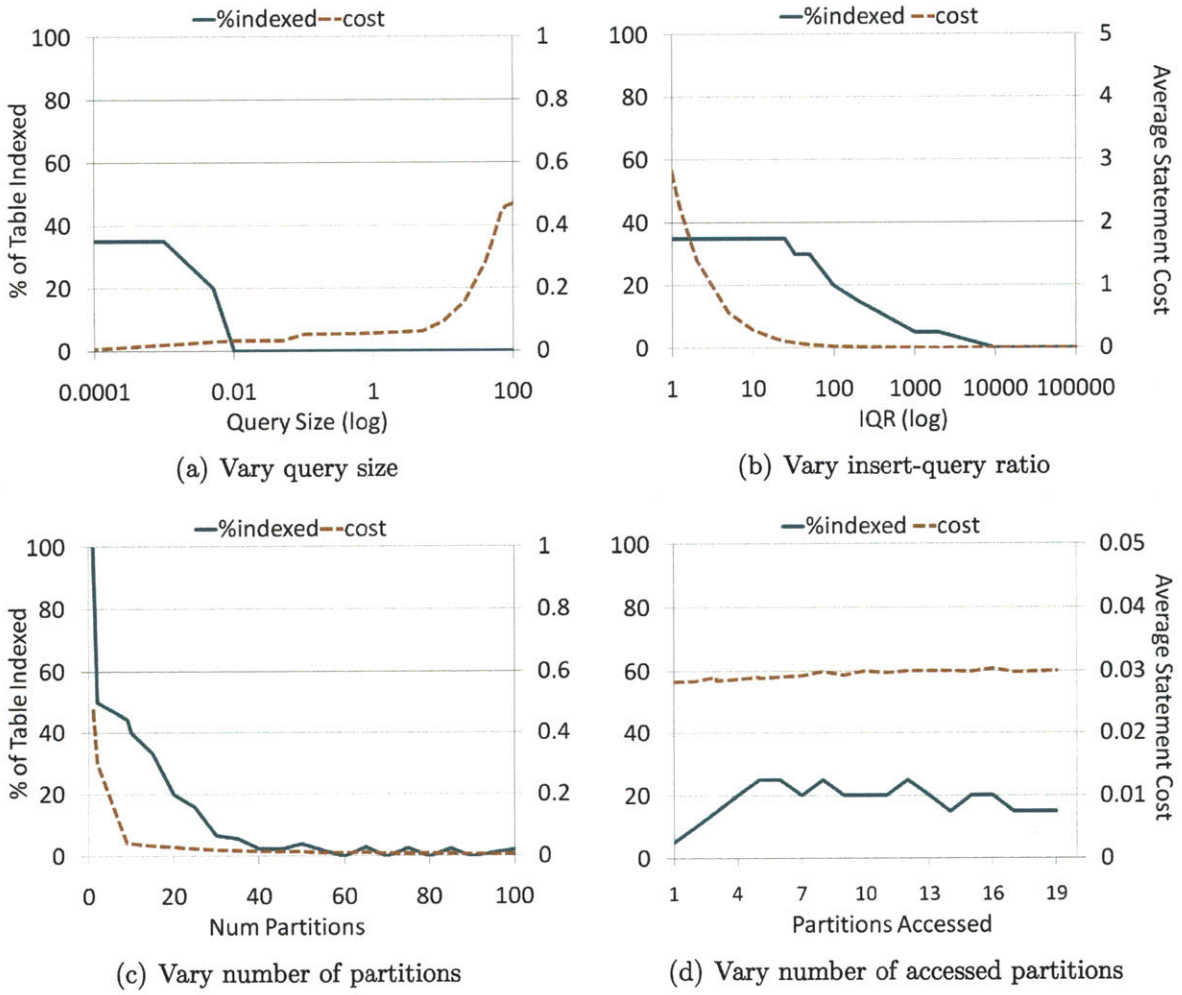
The slopes of the curves represent the workload performance and any repartitioning or indexing costs. $FISP_{10}$ and $SISP_{10}$ have a low initial cost, but quickly outpace $FISP_{50}$ and $SISP_{50}$, respectively, due to poor query performance when accessing larger partitions. However, it is interesting to note that $SISP_{10}$ out-performs the more optimally partitioned $FISP_{50}$ simply by reducing index update costs. Shinobi converges to the same slope as $SISP_{50}$ and initially partitions the table in nearly 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. Most importantly, *Shinobi processes the entire workload before* $FISP_{50}$ *and* $SISP_{50}$ *finish processing the first day.*

Figures 6-4(a) and 6-4(c) show the workload and cumulative costs predicted by the cost model. Although Shinobi scales the expected costs up, the relative differences between the

different strategies are accurately predicted. For example, we accurately predict the cross-over point between Shinobi and $SISP_{50}$ in Figure 6-4(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 50 partitions performing better than 1 or 10 partitions, and with dynamic indexing providing significant overall reductions in insertion costs.

6.2.7 Analyzing Workload Characteristics



(a) Vary query size

(b) Vary insert-query ratio

(c) Vary number of partitions

(d) Vary number of accessed partitions

Figure 6-5: Percentage of table that is indexed and avg. statement cost vs. workload characteristics

In this section, we use the cost model to study how different workload characteristics affect the query and insert performance. Because insert performance is directly related to the size of the indexes (see Appendix ??), the cost model is used to estimate the percentage of the table that is indexed (*%indexed*) and the expected cost per workload statement (*cost*). We use a synthetic workload consisting of 100 queries generated from an exponentially decaying distribution. By generating synthetic queries, we are able to control for a range of system parameters, including 1) *Query Size*, the percentage of the table each query accesses (Default: 1%), 2) *Insert to Query Ratio (IQR)*, the number of insert statements for every select statement (Default: 100), 3) *# Partitions*, the number of equally sized partitions the table is split into (Default: 20), and 4) *Partitions Accessed (PA)*, the number of partitions accessed by one or more queries (Default: 7).

Figure 6-5 shows how the *cost* and *%indexed* curves vary with respect to each of 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 *cost* curve, in seconds.

Figure 6-5(a) varies the query size from 0.0001% to 100% of the table, plotted on a logarithmic scale. The queried partitions are fully indexed for queries smaller than 0.001%. From 0.001% to 0.01%, Shinobi drops indexes on sparsely queried partitions. Above 0.01%, the cost of maintaining indexes exceeds their query benefit and all indexes are dropped. As expected, *cost* increases as more data is accessed, and levels off when the query size exceeds 50% because the table is accesses using full table scans.

Figure 6-5(b) varies the IQR from 1 to 100k, also plotted on a log scale. We can see that the *cost* curve decreases towards the average insert cost as the workload becomes insert dominated. Each query is so expensive because each accesses 30k records through an unclustered index, which will read a large number of pages in the partition. The *%indexed* curve starts at 35%, where all of the queried partitions are indexed, and starts decreasing past IQR=30 because the indexes on sparsely queried partitions are dropped. Shinobi continues to drop indexes until IQR = 10K, at which point none of the partitions are indexed.

Figure 6-5(c) shows that the number of partitions has a large effect on both curves.

The *%indexed* curve decreases because partitioning provides more opportunities to drop indexes on lightly queried partitions. The cost curve decreases sharply due to better insert performance and improved index and sequential scans from accessing smaller partitions.

Figure 6-5(d) varies the number of partitions the workload accesses. The *%indexed* curve increases linearly until $PA = 5$ because the number of partitions increases while the frequency of queries in each partition remains high. As PA continues to grow, the number of queries in some partitions decreases until the number is too low to justify indexing those partitions. The *cost* curve slightly increases with PA due to decreased query locality.

6.3 Multi-Dimensional Workload

In this section, we show that Shinobi is able to effectively optimize tables that are partitioned on multiple partition keys and results in performance improvements consistent with those in the previous section.

The setup and workloads are largely the same as in Section 6.2, so we quickly highlight the important differences in Sections 6.3.1-6.3.3 and report the results in Sections 6.3.4 and 6-8.

6.3.1 Dataset

These experiments use the same Cartel dataset as described in Section 6.2. The values of the lat and lon attributes uniformly fall within the ranges $[35, 45]$ and $[-80, 70]$, respectively, which we define as the dataset boundary. The *centroidlocations* table is indexed and partitioned on the (lat,lon) composite key.

6.3.2 Workloads

Similar to the one-dimensional experiments, we consider a Cartel based workload, $W_{cartel2D}$ and a synthetic workload, $W_{synth2D}$, each consisting of multiple timesteps. Each timestep contains a day's worth of queries and inserts. The inserts are uniformly distributed across

the lat,lon value space. The queries access the data within a region that is 0.1% of the boundary area and are of the form:

$$\text{select * from centroidlocations where}$$
$$[minlat] <= \text{lat and lat} < [maxlat] \text{ and } [minlon] <= \text{lon and lon} < [maxlon]$$

The Cartel workload $W_{cartel2D}$ contains 10 timesteps. For each timestep, we pick a Cartel trace file, compute the access distribution, and sample 100 queries from the distribution.

The synthetic workload $W_{synth2D}$ contains 8 timesteps. The queries in each timestep are sampled from a two-dimensional Gaussian centered around a random point with a standard deviation of 2.5× the query size. The centers of the Gaussians repeat after every 3 timesteps. For example, the distribution in the fourth timestep is the same as the distribution in the first. The reason we use repeating workloads is to additionally illustrate a benefit of partitioning over indexing.

6.3.3 Approaches

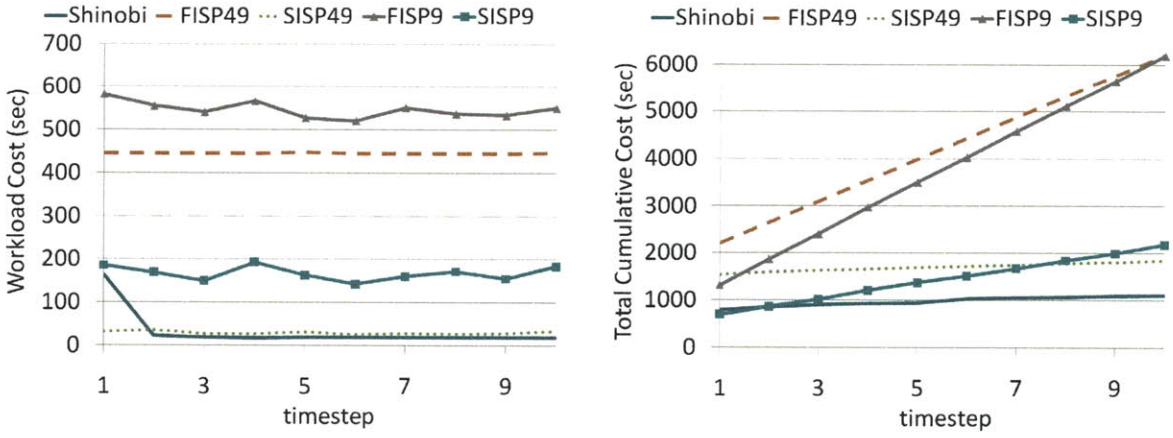
We use approaches that differ along the same dimensions as described in Section 6.2.3. We compare *Shinobi* against full and selective indexing on two static partitions. The static partitions split the table into a 5x5 ($FISP_{25}, SISP_{25}$) or 10x10 ($FISP_{100}, SISP_{100}$) grid.

6.3.4 Cartel Results

We can see that the relative performances of the different approaches in Section 6.2.6 extend to multi-dimensional partitions.

Figure 6-6(a) illustrates the workload-only cost in each timestep. There is a 1.2× improvement after increasing the number of partitions from 9 (3×3) to 49 (7×7) partitions, and a 2.75× and 11× improvement, respectively, after enabling selective indexing. Shinobi quickly adapts to the workload and outperforms the other approaches by the second timestep.

Figure 6-6(b) shows the cumulative workload costs after each timestep. We again see trends similar to Figure 6-4(d). Splitting into 49 partitions is initially very expensive, however



(a) Workload cost per timestep

(b) Cumulative workload cost

Figure 6-6: Shinobi performance on Cartel 2D workload

the cost is amortized across the timesteps and $FISP_{49}, SISP_{49}$ finishes the workload faster than $FISP_9, SISP_9$. Shinobi outperforms the other approaches by having a low initial cost and adapting the table configuration so that it finishes processing the workload before $FISP_{49}, SISP_{49}$ and $FISP_9$ finish processing the first timestep.

Shinobi focuses partition costs on regions that are heavily queried. In Figure 6.3.4, 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. The indexed partitions are accessed often enough to offset the cost of updating the indexes. Note that the indexes are on lat,lon and not on time.

6.3.5 Synthetic Results

In this experiment, we compared $SISP_{25}$, $SISP_{100}$, and *Shinobi* on the synthetic workload $W_{synth2D}$. Figure 6-8(a) plots the workload performance over the 8 timesteps, ignoring partitioning costs. We first note that the relative performance of the approaches match previous experiments, so we instead shift focus and discuss using partitioning to deal with long term workload periodicity.

We can easily see the periodicity of the workload from the $FISP_9$ and $SISP_9$ curves as the queries access new regions of the table; timesteps 1-3 are repeated in 4-6 and 7-

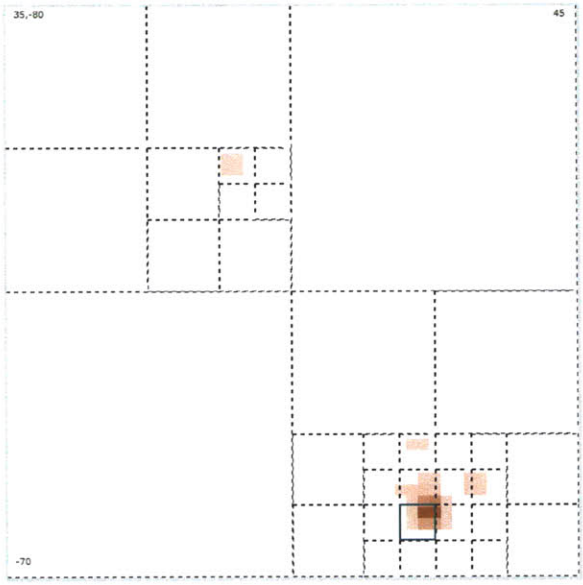
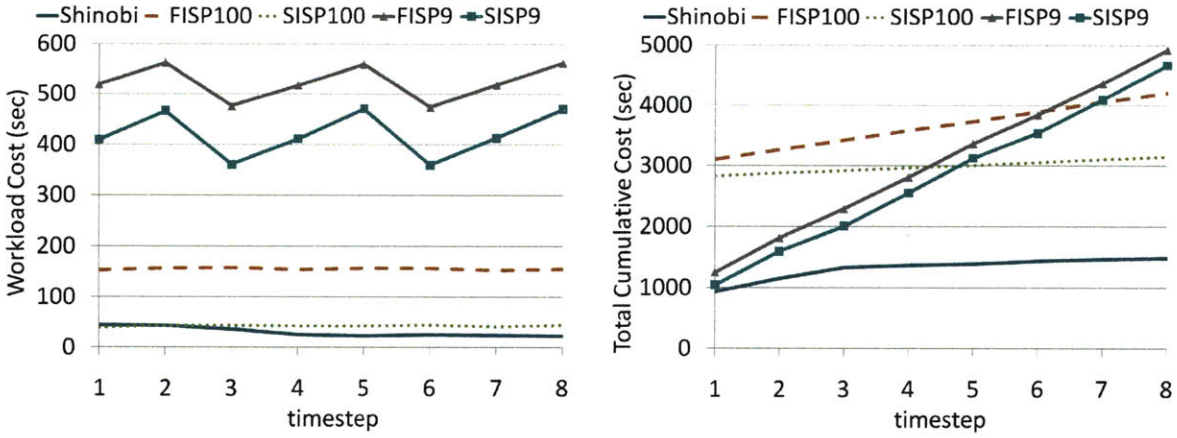


Figure 6-7: Shinobi's 2D partitions and indexes for Nov. 19 workload

8. This highlights one of the downsides of static partitioning – periodic workloads must incur reindexing costs each time the workload shifts. For example, Figure 6-9(a) depicts the workload in timestep 1, which indexes two of the nine partitions. These indexes are dropped when the workload shifts away in timestep 2 (Figure 6-9(b), and then reinstalled in timestep 4.

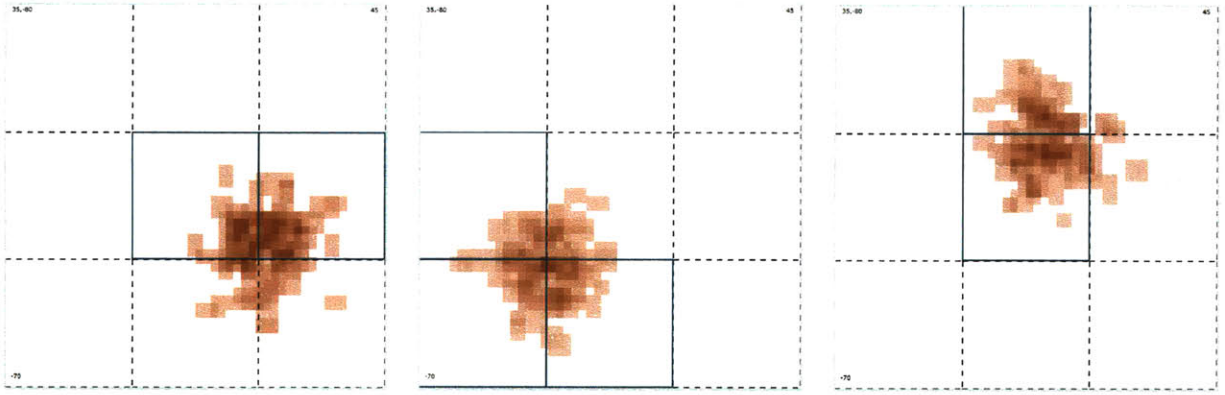
We found that partitioning is an effective way of avoiding reindexing costs. First, a finer partitioned dataset both reduces the number of indexes that must be installed and the cost of indexing a given partition. Second, sequential scan performance may be fast enough that indexes are not needed. Both Shinobi and $SISP_{100}$ do not exhibit this periodicity because they split the table into partitions small enough that they do not need to be indexed. The key difference between the approaches is that Shinobi only partitions the queried regions and uses the cost model to find the optimal partition size.



(a) Workload cost per timestep

(b) Cumulative workload cost

Figure 6-8: Shinobi performance on a synthetic 2D workload



(a) Timesteps 1, 4, and 7

(b) Timesteps 2, 5, and 8

(c) Timesteps 3 and 5

Figure 6-9: Visualization of synthetic workload

6.4 Wikipedia Workload

In this section, we show how Shinobi, coupled with a record shuffling technique can dramatically improve the performance of queries accessing Wikipedia's heavily used *revision* table. These experiments ignore server level caching effects and focus on database performance.

6.4.1 Dataset

Wikipedia contains 11 million articles, and a total of 198 million article revisions, which are stored in three primary tables. The *page* table contains a record for every unique article,

the *revision* table contains a record for each revision of an article, and the *text* table stores blobs containing the text content in each article. The page table has 11.3 million records, comprising 1.01GB of raw data and 0.9GB of indexes. The revision table has 198 million records, 33GB of raw data and 27GB of indexes. In the following set of experiments, we focus on improving access performance to the revision table, which is queried every time an Article is accessed.

The page and revision tables respectively contain a record for every article in Wikipedia, and every revision of each article. The primary fields in the page table are *page_id*, a unique identifier for each article, and *page_latest*, the id of the most recent revision record. The main fields in the revision table are *rev_id*, the auto increment id for each record, and *rev_page*, a reference to the article. Each time an article is edited, a new revision record is created and the *page_latest* field in the corresponding page record is updated to the id of the new revision record.

We call the revision records corresponding to the latest revision of their articles as *latest revision records.*

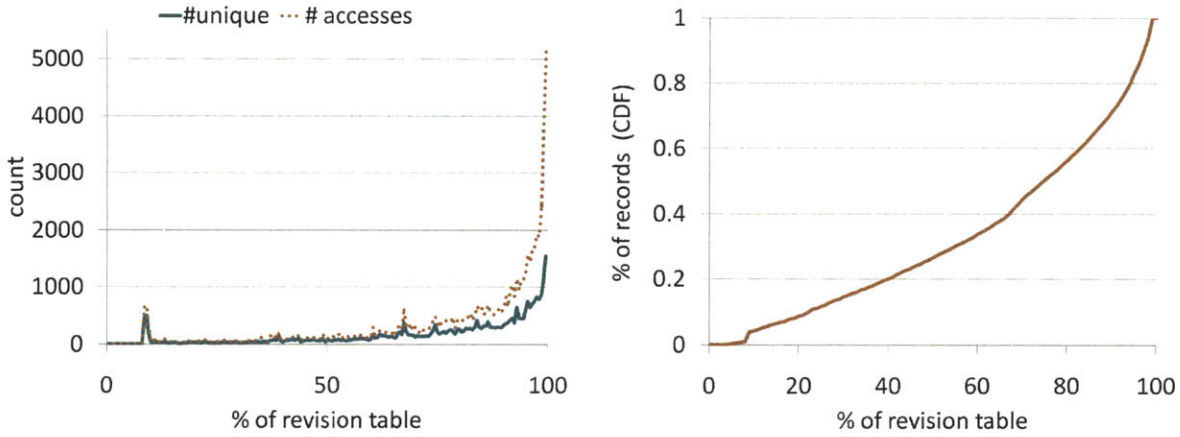
6.4.2 Workload

We used two Apache log files representing 10% of 2 hours of Wikipedia's actual workload. The files contain 5,231,742 HTTP requests, of which 792,544 are requests for an article (the other requests access images, administrator pages and other content not related to the revision table). 908 ($< 0.1\%$) of the requests are edits of an article – for all intents and purposes, the revision table is read-only.

To answer a request, the server first canonicalizes the article name, looks up the id of the latest revision in the page table, and finally uses the revision table to retrieve the id of the appropriate text blob. We log the requests to the revision table and play them back during the experiments.

It is well known that the distribution of article requests follows a zipfian distribution; 50% of the accesses are in the most recent 10% of the table (Figure 6-10(a)). Furthermore, there

is also significant skew in the revision that is accessed – 99.999% of the requests access the latest revision of an article. This skew suggests that the database need only index the latest revision records (10% of the revision table) in order to quickly serve the entire workload.



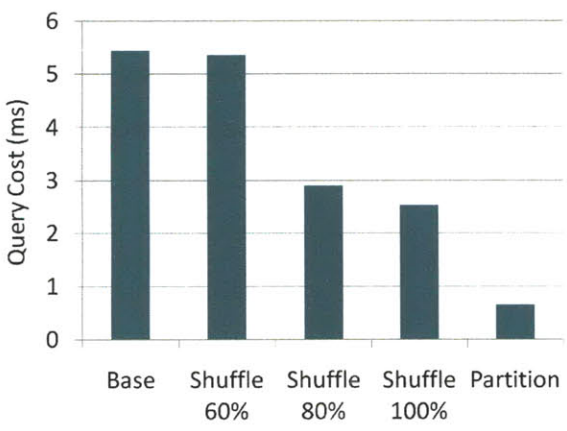
(a) Access frequency of total and unique latest revision records

(b) CDF of latest revision records

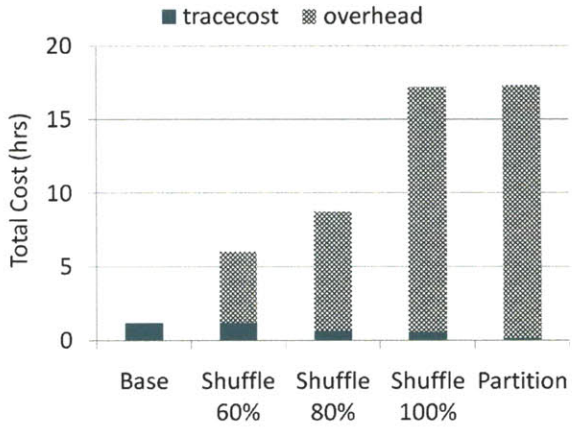
Figure 6-10: Access and data frequency of revision table

Unfortunately, the distribution of records in the revision table does not match the access pattern. Figure 6-10(b) plots the CDF of the number of accesses to each region of the revision table in the 2 hour trace file. The value of the x-axis is the oldest X percent of the table and the y-axis is the total number of requested records within that range. For example, 50% of the latest revision records (5.5 million) are located in the historical 70% of the table. Put another way, 70% of the table needs to be indexed to speed up access to a mere 3% of all the revision records.

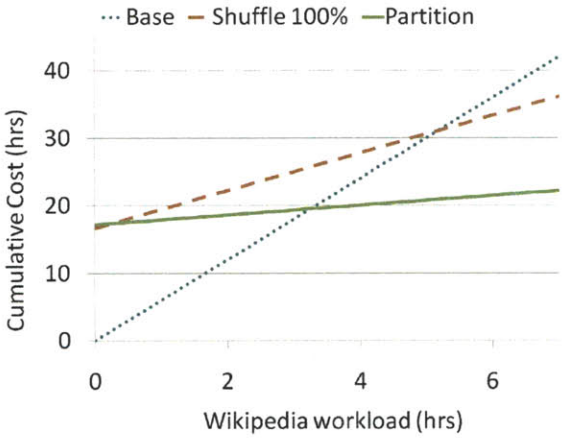
This presents a powerful opportunity to *avoid indexing up to 90% of the table* if we can compact the requests into a semi-contiguous region, possibly reducing the index sizes so that a large fraction fit into memory. In the next subsection, we show that the record shuffling algorithm described in Section 5.5 nearly achieves such gains.



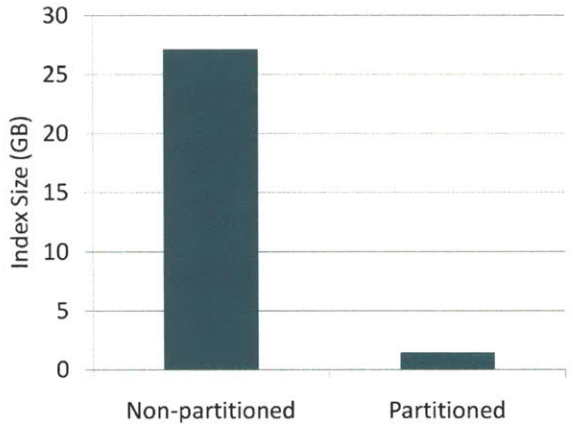
(a) Cost per query



(b) Cumulative trace, shuffling and partitioning costs



(c) Expected trend on full wikipedia trace



(d) Index size

Figure 6-11: Performance of Shinobi and record shuffling on wikipedia workload

6.4.3 Performance Comparison

In this experiment, we compare the benefits of shuffling varying portions of the revision against fully shuffling the table and applying Shinobi's partitioning algorithm. *Base* is the revision table without applying any shuffling or partitioning optimizations. $ShuffleX\%$ shuffles the latest revision records in the oldest X% of the table. *Partition* additionally splits the revision table into historical and latest revision partitions, and drops the indexes on the historical partition.

Figure 6-11(a) shows that after the oldest 80% of the table, or 50% of the latest revision

records, have been shuffled, there is a noticeable performance improvement. *Shuffle*80% and *Shuffle*100% reduce the cost of a query from 5.5 to 2.9 and 2.5 milliseconds. *Partition* further reduces the costs to 0.64 milliseconds per query.

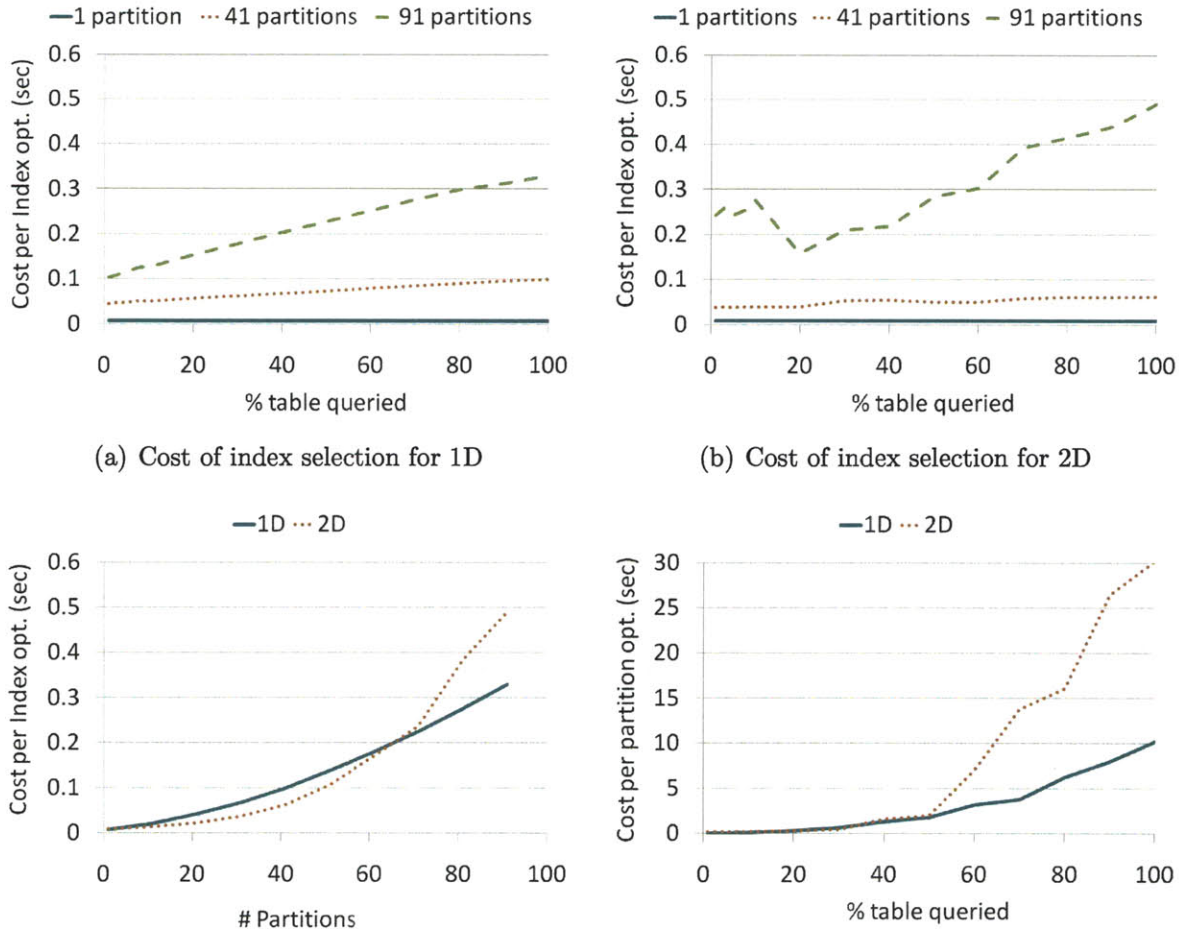
Figure 6-11(b) compares the total time to shuffle the data with the cost of running the trace to completion. The shuffling time is very expensive – nearly 14× the cost of simply running the trace on an unmodified table. Fortunately, the above performance slowdown is illusory. We first note that our trace consists of only *10% of 2 hours* of Wikipedia's typical workload. As shuffling is largely a one-time process, its cost will quickly be amortized. Figure 6-11(c) depicts the expected trend of the cumulative workload and shuffling cost on a full wikipedia trace (100% of the workload). We can see that *Shinobi*100% and *Partition* would out-perform the unmodified table within 5 and 3 hours, respectively. For a website that has been running for over 9 years, we argue that this cost is negligible compared to ability to improve query performance by 8.5× without any hardware investment.

Finally, Figure 6-11(d) illustrates the change in total index sizes from 27.1GB to 1.4GB after Shinobi partitions the table and drops the indexes on the historical partition. The 19× reduction dramatically increases the likelihood that the index is cached in memory.

6.5 Optimization Costs

In this section, we analyze how Shinobi's index selection and partitioning algorithms scale with respect to the workload skew, the number of partitions, and the dimensionality of the partitioning key. These costs correspond to the overhead of running Shinobi's optimization algorithms. The algorithms are executed on a simulated workload and dataset and we only measure the optimization time – queries are not executed. We show that this overhead is acceptable for the skewed workloads that Shinobi is designed for.

The workload distribution represents a set of queries that uniformly access a contiguous sub-range of the table. The size of the sub-range varies from 1% to 100% and each query accesses 0.1% of the data. For each workload, we run the index selector and partitioner for one dimensional and two dimensional partition keys. The index selector is configured with



(a) Cost of index selection for 1D

(b) Cost of index selection for 2D

(c) Index selection cost as # of partitions increases

(d) Partitioning cost as workload accesses larger ranges

Figure 6-12: How index selection and partitioning scale

a number of partitions that varies between 1 and 91. The reported numbers are the means of 5 runs.

Figures 6-12(a) and 6-12(b) show how the performance of the index selector changes as we vary the query range and number of partitions. The cost increases linearly with the size of the queried range, which is expected because the algorithm ignore partitions that are not queried. However, the cost increases super-linearly with respect to the number of partitions. We can see this more clearly in Figure 6-12(c) where the x-axis increases the number of partitions and the y-axis plots the index selection cost for a workload that queries the full table (worst case). The trend corroborates the runtime analysis in Section 5.1 – in the worst

case, the algorithm must run for N iterations, each time computing the benefit of indexing another one of N partitions.

Figure 6-12(d) compares the cost of one and two dimensional partitioning as the workload query range increases. Both curves run in polynomial time, and the 2D curve increases at more than twice the rate of the 1D curve. There are two reasons for this. First, the algorithm partitions the queried regions, thus it creates more partitions if the table accesses more of the table. Second, the cost of index selection, which is called by the partitioner, increases with both the accessed range and the number of partitions. Fortunately, the total cost is below 1 sec when the workload accesses fewer than 30% of the dataset, which aligns with our initial assumptions of a skewed query workload.

In the Cartel and Wikipedia experiments, Shinobi takes less than 1s in each round of optimizations and the cost of running the optimizers does not impact the reported numbers.

Chapter 7

Conclusions

In this thesis we presented Shinobi, a system for partitioning and indexing databases for *skewed query workloads* with 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 insert-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 partitions and indexes, dynamically adjust partitioning and indexing over time and reorder records so that popular records are close together on disk.

Our first set of experiments show dramatic performance improvements on a real-world data set of GPS data and queries from a traffic analysis website, with average performance that is 60× better than an unpartitioned, fully indexed database. Our second set shows that Shinobi, coupled with a record shuffling technique, can achieve more than 8.5× query performance of a real trace on Wikipedia's main table and reduce the size of the table's indexes from 27 to 1.4GB.

Bibliography

[1] http://www.wikipedia.com. http://www.wikipedia.com.

[2] Postgresql 8.1.20 documentation. http://www.postgresql.org/docs/8.1/static/release-8-1.html.

[3] Wikipedia:modelling wikipedia's growth. `http://en.wikipedia.org/wiki/Wikipedia:Modelling_Wikipedias_growth`.

[4] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. In VLDB, 2003.

[5] Devesh Agrawal, Deepak Ganesan, Ramesh K. Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: An optimized index structure for flash devices. PVLDB, 2(1):361–372, 2009.

[6] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, and Vivek Narasayya. Index tuning wizard for microsoft sql server 2000.

[7] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes for sql databases. pages 496–505, 2000.

[8] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In SIGMOD, pages 359–370, New York, NY, USA, 2004. ACM.

[9] S. Babu and J. Widom. Continuous queries over data streams. In SIGMOD, 2001.

[10] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In SIGMOD, pages 128–136, New York, NY, USA, 1982. ACM.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In CIDR, 2003.

[12] Surajit Chaudhuri. An overview of query optimization in relational systems. In PODS, pages 34–43, New York, NY, USA, 1998. ACM.

[13] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In VLDB, pages 146–155, 1997.

[14] Surajit Chaudhuri and Vivek Narasayya. Autoadmin "what-if" index analysis utility. 27(2):367–378, 1998.

[15] Douglas Comer. Ubiquitous b-tree. In ACM Computing Surveys, volume 11, pages 121–137, New York, NY, USA, 1979. ACM.

[16] P. Cudre-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In ICDE, 2010.

[17] Alan Demers, Johannes Gehrke, and Biswanath P. Cayuga: A general purpose event monitoring system. In CIDR, pages 412–422, 2007.

[18] Raphael Finkel and J.L. Bentley. Quad trees: A data structure for retrieval on composite keys. In Acta Informatica, 1974.

[19] Goetz Graefe. Partitioned b-trees - a user's guide. In BTW, pages 668–671, 2003.

[20] Goetz Graefe. Sorting and indexing with partitioned b-trees. In CIDR, 2003.

[21] Goetz Graefe. Write-optimized b-trees. In VLDB, pages 672–683, 2004.

[22] Goetz Graefel and Kuno Harumi. Adaptive indexing for relational keys. In SMDB, 2010.

[23] Goetz Graefel and Kuno Harumi. Self selecting, self-tuning, incrementally optimized indexes. In EDBT, 2010.

[24] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A.K. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In 4th ACM SenSys, Boulder, CO, November 2006.

[25] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In SIGMOD, pages 297–308, New York, NY, USA, 2009. ACM.

[26] M. Jarke and J. Koch. Query optimization in database systems. In ACM Computing Surveys, 1984.

[27] Christopher Jermaine. A novel index supporting high volume data warehouse insertions. In VLDB, pages 235–246, 1999.

[28] Martin L. Kersten and Stefan Manegold. Cracking the database store. In CIDR, pages 213–224, 2005.

[29] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. Zdonik. CORADD: Correlation aware database designer for materialized views and indexes. In PVLDB, 2010.

[30] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). Acta Inf., 33(4):351–385, 1996.

[31] Stratos Papadomanolakis and Anastassia Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In SSDBM, page 383, Washington, DC, USA, 2004. IEEE Computer Society.

[32] P. Seshadri and A. Swami. Generalized partial indexes. In ICDE, 1995.

[33] M. Stonebraker. The case for partial indexes. In VLDB, 1987.

[34] Gary Valentin, Michael Zuliani, and Daniel C. Zilio. Db2 advisor: An optimizer smart enough to recommend its own indexes. In ICDE, pages 101–110, 2000.

[35] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In SIGMOD, 2006.