Adaptive partitioning and indexing for raw data querying

Matthaios Olma

DIAS, I&C, EPFL

***Abstract*—Traditional database management systems approach to data analytics assumes that the input would be loaded within the DBMS, and then queried upon. However, data analytics depend on the interaction with the data analyst and as data collections grow larger and larger, data loading acts as a bottleneck and it incurs significant *data-to-query* delay.**

In this paper, we examine the NoDB paradigm, which proposes querying data *in situ*. We describe the structures it introduces to overcome the limitations of having to access raw files repeatedly, its adaptive nature, and see how it is manifested in a system extending a traditional DBMS. To address shortcomings of query execution in NoDB, we examine a parallelized approach to in situ query processing. The efficient design of concurrent execution may further minimize the overheads of raw data accesses as well as benefit the query execution. In addition, affected by the sheer amounts of data that need to be handled, and the dynamic workloads which are common in data exploration scenarios, we present an adaptive partitioning and indexing system for relational tables which may be applied accordingly to fit the raw data querying paradigm.

Finally, inspired by the previous approaches, we briefly present our vision for a lightweight database service which allows users to seamlessly pose interactive queries over raw files of unlimited size while minimizing the execution time.

Proposal submitted to committee: May 28th, 2013; Candidacy exam date: June 4th, 2014; Candidacy exam committee: Prof. Rachid Guerraoui, Prof. Anastasia Ailamaki, Dr. Edouard Bugnion.

This research plan has been approved:

Date: _____

Doctoral candidate: _____
(M. Olma) (name and signature)

Thesis director: _____
(A. Ailamaki) (name and signature)

Thesis co-director: _____
(if applicable) (name and signature)

Doct. prog. director: _____
(B. Falsafi) (signature)

EDIC-ru/05.05.2009

***Index Terms*—In situ querying, Query optimization, Data analysis, Parallelization, Partitioning, Indexes**

I. Introduction

Experts from various fields, ranging from domain scientists to data warehouse analysts, attempt to gain knowledge by going through newly acquired data and combining them with existing information. Through this process, and by exploiting a number of different data sources they try to expose previously unknown patterns of interest.

The continuously increasing amounts of daily produced data, open up multiple possibilities for extracting useful information via its exploration, but at the same time introduce major challenges in data management.

In order to extract information from the bulks of produced data one has to employ techniques like data exploration. In data exploration cases, in order to gain insight through large amounts of never-seen-before data, one has to follow a query-based approach where knowledge over the data advances through querying and leads eventually to gathering previously unknown information.

Despite the increase in computing resources, due to the volume of the data produced and the effort needed to analyze it, the data cannot be leveraged. Deriving new information from data is a time-consuming and complex process. Besides the sheer amount of data produced, bottleneck to this process is also the complexity of the data and the variety of formats the data is stored in.

The traditional approach for data exploration follows the ETL (Extract, Transform, Load) scheme thus requiring the full loading of datasets into a DBMS. The scenarios for simple data exploration have indicated that no useful information resides in a batch of data, meaning that it may be ignored. Thus, full loading of data into a DBMS, being a costly process, introduces a major investment which may not be leveraged.

The primary reason for the presence of these time-consuming tasks is the requirement of database systems to always have complete control over the data. Therefore, the impedance mismatch between the raw data and the query engine has traditionally been resolved by converting raw data to an internal, database-specific representation and loading it in a database system. Then, queries would take place using a query engine which has been heavily optimized to offer high performance when operating over the internal format.

But this impedance mismatch with the data format, is only one dimension of the problem. As the workload tends to be

very dynamic it is only logical that DBMS should adapt to it as well. In this context, we argue that by incorporating into a DBMS the capabilities to adapt to the data and the workload while exploiting both computational and storage resources in an efficient way, the execution of analytical queries will be more efficient.

The rest of this paper is organized as follows: Section II introduces the NoDB philosophy, which proposes querying data *in situ* instead of loading them in a traditional DBMS. In Section III we examine a parallel solution to in situ query processing with a different approach to loading. In Section IV we present adaptive partitioning and indexing techniques over relational data which can help further with the adaptation of the execution to the workload. Finally, in Section V we briefly present our research plan, outlining our approach for a lightweight, data- and query-adaptive database service over raw data.

II. Raw data query paradigm

The authors of [1] introduce the "NoDB philosophy". Motivated by the trends in scientific data analysis they recognize that to make data exploration efficient, the processes have to be more interactive and in order to succeed the data-to-query time has to be reduced. Traditionally, a data analysis task demands that data has to be loaded into a DBMS, despite that only a small part of the data would prove relevant. NoDB goal is to remove this significant bottleneck and provide a load-free data exploration experience.

In order to achieve its goal, NoDB executes queries straight on the "raw" data files and to make this as efficient as possible, treats raw data as a first-class citizen of a DBMS. The scan operators of a NoDB system are to be able to handle not only the binary data format that is understandable by the database, but also raw data. In addition, specialized data structures facilitate raw data access by providing indexing support over raw data. This deep integration of raw data support differentiates NoDB from industrial efforts such as the ones from Oracle and MySQL. These two systems offer support for querying files external to a database, but the approach followed by them leads to poor performance, as files are treated as an external entity, without indexing support, in-database caching, or statistics gathering. NoDB opts instead to treat "raw" data as a first-class citizen of the DBMS.

To prove the viability of the NoDB philosophy the authors implemented a system prototype. The system, nick-named PostgresRAW, has been built by overriding the scan operators of the PostgreSQL row store. It provides support for querying raw data stored in comma-separated value (CSV) files.

A. Reducing in situ access overheads

As a result of removal of loading functionality a number of significant limitations that have to be taken into account. Through loading, databases gained control over the data and thus introduced metadata that improved query execution. In the case of in situ access every time a query has to be executed the file has to be scanned to determine the tuples present (i.e., find the end of line for each row), then the tuple fields need to be tokenized based on the delimiter used, and finally the fields required need to be converted to their original data types. Only then the tuples can be evaluated against the query.

The authors to overcome the overheads of raw data file scan implemented a variety of techniques. A first step includes "selective tokenizing" based on which the tuple tokenizing stops as soon as the attributes necessary to answer a query have been found. In order to reduce parsing costs, PostgresRAW can stop converting attributes of a tuple as soon as this tuple does not satisfy some selection predicate, assuming there is a WHERE clause predicate. Furthermore, PostgresRAW performs selective tuple formation, creating tuples comprising only the attributes required by the query. All in all, by using these techniques, CPU processing costs are reduced, although I/O costs are not affected.

Furthermore, PostgresRAW uses an auxiliary structure that stores positional information of attributes. This structure, called "positional map" aims to reduce the file scan and as a result to provide faster data retrieval. The positional map is dynamically populated during query execution. Whenever an attribute is retrieved from a raw file, its position is inserted in the positional map. Subsequent queries requesting this attribute, can use this information to directly jump to the appropriate position, thus reducing parsing and tokenizing costs. Even requests for nearby attributes can be serviced by incrementally parsing from the known positions, so that scanning tuples from their beginning is avoided.

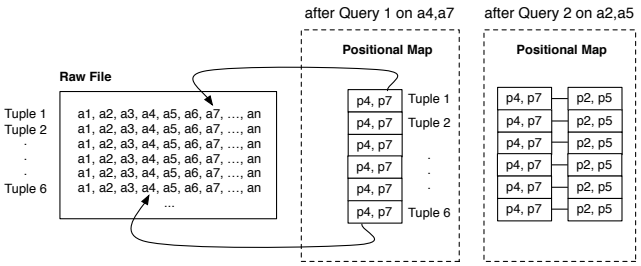

Fig. 1: Example of a positional map

An example of a positional map instance is depicted in Figure 1. Its structure and contents fully adapt to the query workload.

In order to further reduce data file access cost PostgresRAW is also reinforced by the presence of data caches. These caches aim to reduce parsing, tokenizing as well as conversion to binary costs.

Finally, PostgresRAW adaptively generates statistics on the attributes of the file that are touched by queries. The more "popular" some fields are, the more statistics will be gathered for them.

B. NoDB performance evaluation

For a NoDB system to be a viable alternative, it needs to offer performance comparable to that of a traditional DBMS. To this end, and to demonstrate the impact of the techniques proposed, PostgresRAW is evaluated against a number of existing solutions.

Experiments are performed using CSV input consisting of 7.5×10^6 tuples, with each tuple containing 150 integer fields. A subset of the results is depicted in Figure 2. Approaches like MySQL and DBMSX external tables which naively re-visit raw files in every access cannot be considered a competitive option. However, PostgresRAW even manages to be have performance competitive to state-of-the-art DBMS while allowing rapid access to data. The extra cost of having to access raw data is amortized among the query sequence, and is reduced as the structures used are adaptively refined.

An additional experiment providing useful insights is depicted in Figure 3. In this scenario, PostgresRAW is compared with systems executing queries over pre-loaded data, all starting with a cold cache. The projectivity of the query is reduced between runs, while keeping selectivity at 100%. After the initial query, which is a worst-case scenario for NoDB systems (as the entire file needs to be scanned), PostgresRAW is competitive with the other systems. Actually, as projectivity is reduced, CPU processing in the case of PostgresRAW is reduced more rapidly than in the case of PostgreSQL. This is due to the fact that only necessary attributes are brought in the CPU caches, as mentioned in Section II-A.

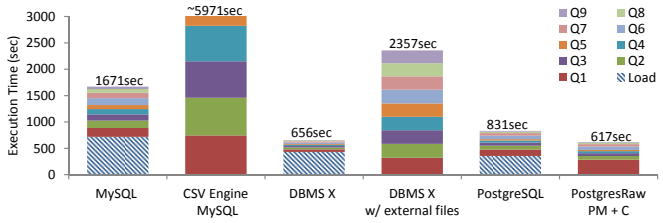

Fig. 2: Comparison of PostgresRAW with other DBMS

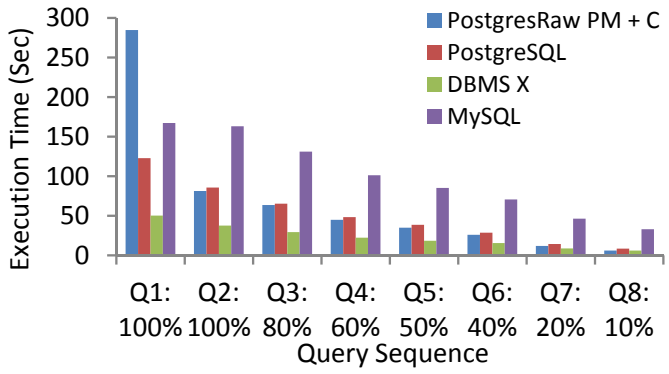

Fig. 3: Comparison of PostgresRAW with other DBMS as a function of projectivity

C. Summary

Summing up so far, PostgresRAW demonstrates a viable alternative for data analytics workloads offering comparable performance to commercial DBMS without incurring the cost of loading.

III. Parallel raw data processing

Ideally a solution operating over raw data would not be restricted to a sequential execution. The sheer amounts of analytics workloads dictate scaling up and parallelizing the execution.

In "NoDB" raw data processing bottlenecks were discovered and multiple approaches (i.e. Selective tokenizing, parsing and data type conversion) to overcome them were introduced. These operations though, are dependent on each other only on the data they get as input. By exploiting this knowledge and being motivated by evolution of multi core processors, the authors [2] use multi-threaded execution to parallelize operations and thus enable a more efficient query execution over raw data.

A system prototype named SCANRAW, has been implemented. This system has been built by appropriately introducing the parallel execution as well as some additional functionality into the "NoDB" philosophy. Apart from the parallel execution of separate operations, SCANRAW takes a different approach to loading. Instead of fully removing the overhead of writing to disk, SCANRAW introduces "Speculative loading" approach, based on which used data will be written onto disk when the storage device is underutilized, (i.e., execution is CPU bounded).

A. SCANRAW architecture

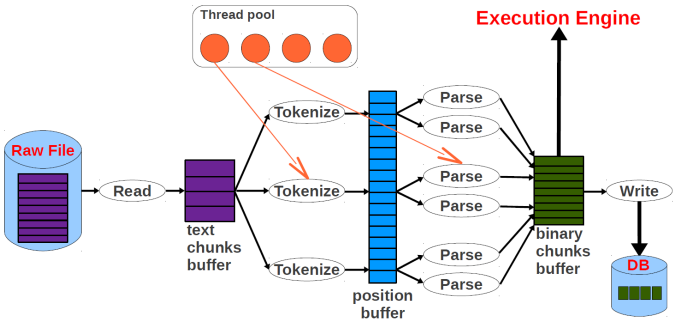

Fig. 4: Architecture of SCANRAW

Figure 4 depicts the high level architecture of SCANRAW. Primarily, SCANRAW acts as a scan operator and reads data from the raw data files. The read operation takes place in chunks, which is a collection of subsequent tuples. After being read, the data is tokenized and parsed into the tuple representation that can be processed by the execution engine. Both tokenization and parsing are as well executed in chunks of tuples.

As seen in Figure 4 multiple tokenize and parse stages are present. Each of them operates on different portion of the data in parallel, thus the execution of each one is independent to the rest. A very important part of the architecture is the scheduling of these stages and is managed by a separate component, the scheduler. The scheduler controls a thread pool and assigns threads to stages dynamically at runtime. The stages in the SCANRAW pipeline act as producers and consumers that move chunks of data between buffers. The entire process is regulated by the size of the buffers which is determined based on memory availability.

Considering the buffers, the *text chunks buffer* contains text fragments read from the raw file. The file is logically split into

horizontal portions containing a sequence of lines, i.e., chunks. The *position buffer* between tokenize and parse contains the text chunks read from the file and the corresponding field starting positions computed in tokenize. Finally, *binary chunks buffer* contains the binary representation of the chunks. This is the processing representation used in the execution engine as well as the format in which data is stored inside the database. In binary format, tuples are vertically partitioned along columns represented as arrays in memory. When written to disk, each column is assigned an independent set of pages which can be directly mapped into the in-memory array representation. It is important to emphasize that not all the columns in a table have to be present in a binary chunk.

B. Speculative loading

At the end of parsing, data converted to binary is loaded in memory. Multiple paths can be taken at this point. In commercial databases with raw data querying functionality (”External tables”), data is passed to the execution engine for query processing and discarded afterwards. In NoDB, data is kept in memory for subsequent processing and smart caching policies are incorporated for optimal memory utilization. SCANRAW implements speculative loading. In standard database loading, data is first written to disk and only then query processing can begin. SCANRAW on the other hand decides adaptively at runtime (1) what data to load, (2) how much, and (3) when to load it, while maintaining optimal query execution performance. These decisions are taken dynamically by the scheduler. Since the scheduler monitors the utilization of the buffers and assigns worker threads for task execution, it can identify when READ is blocked and enable WRITE accordingly.

The ultimate goal of speculative loading is to be able to take advantage in the future the already invested time into parsing and tokenization but it is paramount that the disk access have to be carefully synchronized in order to minimize the interference since READ and WRITE contend for I/O throughput. Eventually depending on the policy taken by the scheduler, SCANRAW could be either an ”external table” scan operator, a ”NoDB” type operator with only intermediate caches or to degenerate into a parallel Extract-Transform-Load (ETL) operator.

C. Evaluation

The authors validate the impact of their contributions namely, (1) the introduction of multi-threaded execution and (2) the incorporation of speculative loading, with series of experiments. The experiments are conducted on a server with 2 AMD Opteron 6128 series 8-core processors (64 bit) and 40 GB of RAM using a CSV file with 2^{26} rows and 64 columns with total size 40GB.

In Figure 5 the different variations of SCANRAW are evaluated against configurations with varying number of worker threads. The query run is a summation over all columns and rows. The ”Load & Process” initially loads all data and subsequently executes the query thus incorporating a major static cost. The ”External Tables” approach queries straight over

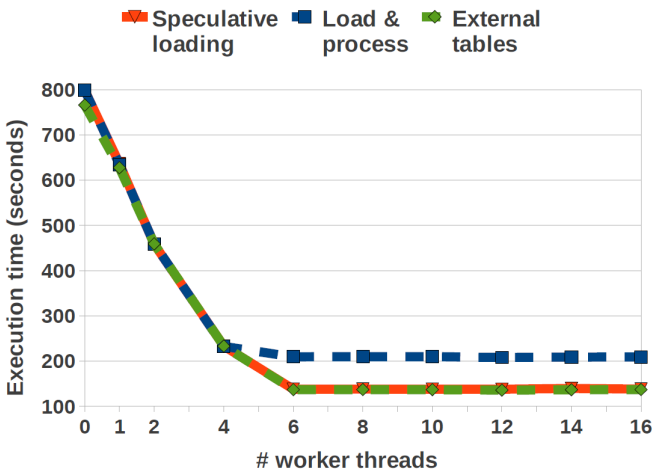

Fig. 5: Execution time against parallelization

the raw data files. The ”Speculative loading” approach loads data when the storage device is idle. In the experiment the execution time level-off beyond 6 workers. The reason for this is that processing becomes I/O-bound. Increasing the number of worker threads does not improve performance anymore. As expected, loading all data during query processing increases the execution time. Before leveling-off the execution time of all three configurations are similar because as processing is CPU-bound and due to parallelism SCANRAW manages to overlap conversion to binary and loading into the database completely. Essentially, loading comes for free since the disk is underutilized. All the unique SCANRAW features super-scalar pipeline, asynchronous threads, dynamic scheduling combine together to make loading and processing as efficient as external tables.

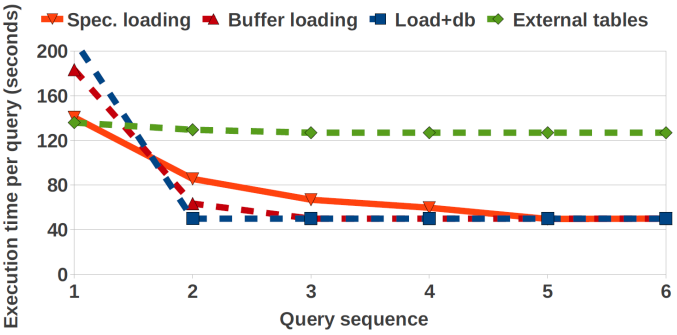

Fig. 6: Execution time for a sequence of queries

In Figure 6 is depicted the execution time of the different approaches for a sequence of 6 same queries (identical queries with the previous experiments).

The execution time for external tables is almost constant. Data are always read from the raw file, tokenized, and parsed before being passed to the execution engine. Considering database execution, the first query incurs the whole cost of loading thus starting from the second query the execution time is constant. Comparing external tables to database execution the database is considerably faster than external tables by a factor of 2.5. Buffered loading simply inserts all chunks that

are expelled from the cache to the database thus distributes the loading time over the first two queries since not all data fit in memory. As a result, there is a decrease in runtime for the first query when compared to standard loading. For the second query though, execution time is larger. Considering the execution speculative loading, for the first query it executes identically as external tables but after a number of queries it converges to the database execution time which is the minimun it can achieve.

D. Summary

SCANRAW demonstrated that parallelizing of tokenization and parsing of queries over raw data can provide considerable enhancements in their performance and at the same time can provide the resources to hide database loading behind query execution without hurting query performance.

IV. Exploit adaptive partitioning and indexing

In many data-intensive applications the velocity with which data is produced is considerably higher than the rate that users access it. Such growing datasets lead to ever-increasing space and performance overheads for maintaining and accessing indexes. At the same time, the skew over popular and not popular data access is considerable. Motivated by these observations the authors of [6] designed Shinobi, a system which uses horizontal partitioning as a mechanism for improving query performance in a relational database by clustering the physical data, and increasing insert performance by only indexing data that is frequently accessed.

In order to partition efficiently and provide fine-grained indexing, thus improving the performance of skewed workloads, Shinobi uses three key ideas:

1) it partitions tables, so that regions of the table that are frequently queried together are stored together, separate from regions that are infrequently queried.
2) it selectively indexes these regions, creating indexes on partitions that are queried frequently, and omitting indexes for regions that are updated but queried infrequently.
3) it dynamically adjusts the partitions and indexes to account for changes in the workload.

A. Architecture

Shinobi acts as an intermediary between the workload and a database system. It provides both functionality to statically partition and index optimally the dataset based on some initial data, and the capability to continuously adapt to the workload.

The input Shinobi gets is: (1) a list of attributes each table is to be partitioned on, (2) a set of indexes to install on the table, and (3) a set of queries and inserts that apply to the table. The indexes to be used may be provided by a database administrator or database tuner. Based on the initial data and through monitoring Shinobi finds an optimal set of non-overlapping range partitions and chooses indexes for each partition (together called ”table configuration”) to maximize workload performance.

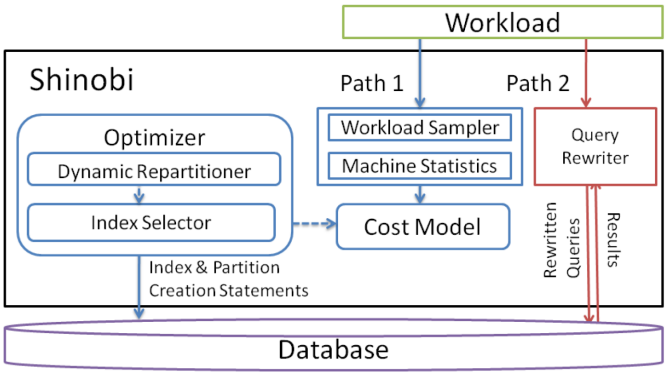

Fig. 7: Shinobi architecture

The architecture of Shinobi is depicted on Figure 7. It depicts in detail the execution of Shinobi. The workload follows two major paths. Through Path 1 Shinobi samples incoming SQL statements and updates workload statistics for the Cost Model. The Optimizer uses the cost model to (re)optimize the table configuration. Through Path 2 Shinobi parses queries using the Query Rewriter. Since changes may have happened in the database schema by Shinobi queries referring to partitioned attributes have to be rewritten and redirected to the relevant partitions. Queries without such predicates are directed to all partitions.

Considering sampling, the Workload Sampler reads recent SQL statements from the query stream and computes workload characteristics such as the insert to query ratio, and the query intensity of different regions of the table. Similarly, the Machine Statistics component estimates capabilities of the physical device as well as database performance information. Physical statistics include RAM size and disk performance, while database statistics include append costs, insert costs, and typical query costs.

Based on these statistics and the defined cost model the expected statement cost for a workload is calculated. 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.

Finally, the Index Selector and Dynamic Repartitioner components optimize the table configuration based on the results of the cost model. More specifically, they choose the most efficient partitioning scheme and the most efficient set of indexes to install on each partition of a table.

B. Cost model

The goal of the cost model is to accurately order the query and update performance of different table configurations rather than exactly estimate the expected cost. In order to succeed Shinobi cost model tries to predict:

1) The average cost per query.
2) The cost to re-partition and re-index a table.
3) The overall benefit to switch to a new ”table configuration”.

Based on the calculation of these metrics the decisions considering the table configurations are taken.

1) Query Cost Model: As the goal of this system is to be efficient even in update-intensive workloads, the cost model tries to incorporate both query and insert costs. It takes into consideration both clustered and unclustered indexes and machine statistics such as database copy speed.

2) Repartition Cost Model: The repartitioning cost model estimates the cost to switch from one table configuration to another. It takes as input the existing configuration and outputs along with the new configuration, the cost of the transformation to the new configuration (new partitions and indexes). The cost is separated into two major sections, the *partition cost* and the *index cost*.

3) Workload Cost Model: To calculate the total benefit that a table configuration transformation can provide, the benefit for a specific workload has to be initially calculated. Based on past statistics, Shinobi estimates the lifetime of a workload. Based on the lifetime, it calculates the average query cost for the current and ”new” table configurations (as described earlier) and multiplies it by the estimated number of queries to come. To make a decision though, the investment into the transformation for the new configuration has to be taken into consideration as well. Thus, the total benefit is calculated by subtracting from the benefit the repartitioning/re-indexing cost. It is clear that if the workload is longer or the benefits from the transformation are larger this scheme allows for more considerable investments into repartitioning/re-indexing.

C. Evaluation

The authors of Shinobi provided extensive experimental evaluation of the system against a number of workloads and datasets. Shinobi has to prove that it enhances query performance when using a realistic workload. To this end, and to demonstrate the impact of the cost model, Shinobi is tested against the real Cartel workload (Wcartel) [3] that contains the positions of vehicles in the Boston area.

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

The workload contains 10 timesteps. Each timestep has 100 queries and 360 inserts per query (36k inserts/timestep). The comparison is done among approaches that differ along two dimensions: index selection technique and partitioning type. Regarding indexing, Full Indexing (FI) indexes all of the data in the table, and Selective Indexing (SI) uses the cost model to only create beneficial indexes. Regarding partitioning, Static Partitioning (SPN) partitions the table into N equally sized partitions, and Optimized Partitioning (OP) finds the optimal partitioning based on the cost model described. The approaches tested include:

- a fully indexed table (FISI1)
- full and selective indexing on a table statically partitioned into N partitions (FISPN, SISPN)
- selective indexing on a dynamically partitioned table (SIOP or Shinobi)

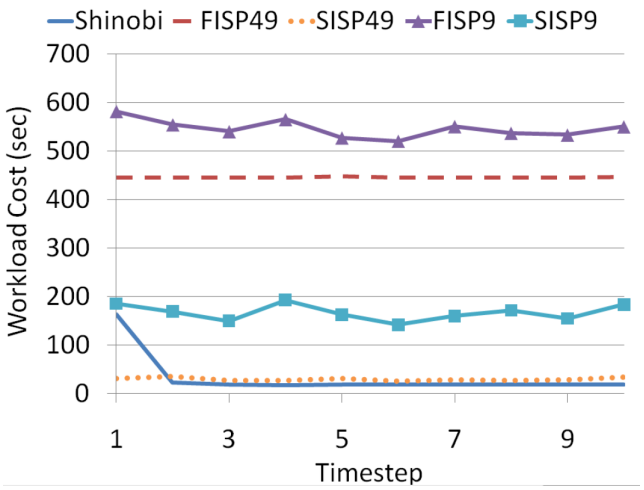

Fig. 8: Cartel workload performance

The results are depicted in Figure 8. Primarily considering partitioning, the FISP9/49 curves illustrate the effectiveness of statically partitioning the table into 9 and 49 partitions respectively. Increasing the number of partitions from 1 to 9 and 49 reduces the select query costs by over 3× and 4×, respectively. Considering selective indexing, as it only creates indexes on heavily queried partitions, and reduces insert costs for SISP9/49 by 7× and 21× respectively. Shinobi performs as well as SISP49; the higher initial cost is because Shinobi has not collected enough data yet and thus invests to much cheaper partitioning.

D. Summary

Partitioning significantly reduces query costs when the dataset is not clustered on the partition keys, whereas selective indexing can dramatically reduce the index size, and correspondingly the index costs, even for clustered datasets. By monitoring the workload and introducing a cost model, Shinobi targets dramatic performance improvements over real-world workloads with average performance that is 60× better than an unpartitioned, fully indexed database.

V. Research Proposal: An Adaptive Query Service for in Situ Querying

As increasingly more companies are basing their business on the results of data analysis and data is generated at an increasing rate, the task to provide to data analysts the tools to efficiently handle their data has become work of extreme importance. A novel approach to offer a scalable and accessible solution to empower data scientists to leverage their data is needed.

From the work described, NoDB [1], introduces in situ query processing and in order to minimize the overhead of raw data file access incorporates various optimization techniques. Motivated by the sheer amount of data, SCANRAW [2] improves over NoDB considering parallel execution. Finally

Shinobi [6], targets at adapting to the workload as much as possible by gaining knowledge over the dataset exploiting this knowledge to efficiently partition and index accordingly.

Inspired by these approaches we propose a lightweight in-situ database management system that needs neither loading nor tuning. This system would adaptively populate its metadata and tune itself depending on the upcoming queries while querying over raw data files. The system takes advantage in an optimal way the existing resources by scaling up and out. By collecting statistics over the execution the system will choose the optimal partitioning schemes paired with specialized indexes. No preparation or loading data would be necessary. By minimizing data-to-query time, data scientists will be empowered to explore their data independent of its volume, variety or complexity.

From a performance perspective the results presented by NoDB are encouraging and also motivated by the fact that prospective benefits from applying partitioning and adaptive indexing will be even greater for raw data files compared to traditional DBMS, due to the avoided disk scans, one can be confident over the efficiency of such an approach.

In order to implement such a system though has a number of challenges. Most partitioning approaches on relational data, incorporate physical restructuring of the data which is not permitted in in situ query processing. At the same time the execution should avoid duplicating data as much as possible. In addition any decisions over the data have to be made as early as possible (first scan), on the one hand to exploit as early as possible the opportunity of speed-up and on the other hand as indexing data incurs a considerable overhead over data scans. All these allow only creating data structures that manage meta-data. The same principles apply for indexes which will have to be designed in accordance to the physical infrastructure and taking advantage the hierarchy of storage devices while providing the lowest insertion overhead, consume minimal space and offer the highest possible performance.

The system ideally would be provided as a library that can be easily incorporated to any data analytics platform providing efficient access to data without requirements for preloading.

All in all, we envision of a database system that through avoiding data loading gains elasticity without losing in performance. This system will be able to adapt to any workload, despite the size of input or complexity of the data. Through this adaptation the cost of execution will be the minimal possible given the physical constraints of technological equipment.

REFERENCES

[1] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.

[2] Yu Cheng and Rusu Florin. Parallel In-Situ Data Processing with Speculative Loading. In *SIGMOD*, 2014.

[3] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. Cartel: a distributed mobile sensor computing system. In *In 4th ACM SenSys*, pages 125–138, 2006.

[4] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, 2011.

[5] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *CIDR*, 2007.

[6] Eugene Wu and Samuel Madden. Partitioning techniques for fine-grained indexing. 2011.