Adaptive Caching in Big SQL using the HDFS Cache

Avrilia Floratou[1], Nimrod Megiddo[1], Navneet Potti[2], Fatma Özcan[1],
Uday Kale[1], Jan Schmitz-Hermes[1]

[1]*IBM* avrilia.floratou@gmail.com, {megiddo, fozcan, udayk}@us.ibm.com, jan.schmitz-hermes@de.ibm.com
[2]*University of Wisconsin-Madison* nav@cs.wisc.edu

Abstract

The memory and storage hierarchy in database systems is currently undergoing a radical evolution in the context of Big Data systems. SQL-on-Hadoop systems share data with other applications in the Big Data ecosystem by storing their data in HDFS, using open file formats. However, they do not provide automatic caching mechanisms for storing data in memory. In this paper, we describe the architecture of IBM Big SQL and its use of the HDFS cache as an alternative to the traditional buffer pool, allowing in-memory data to be shared with other Big Data applications. We design novel adaptive caching algorithms for Big SQL tailored to the challenges of such an external cache scenario. Our experimental evaluation shows that *only* our adaptive algorithms perform well for diverse workload characteristics, and are able to adapt to evolving data access patterns. Finally, we discuss our experiences in addressing the new challenges imposed by external caching and summarize our insights about how to direct ongoing architectural evolution of external caching mechanisms.

Categories and Subject Descriptors H.2.4 [*Database management*]: Parallel databases

Keywords SQL-on-Hadoop, HDFS Caching

1. Introduction

Big Data platforms such as Hadoop and YARN enable enterprises to centralize and share their data among multiple data processing frameworks and applications, including relational databases, machine learning, graph and streaming analytics. The data is often stored in open HDFS data formats and ownership is shared between these frameworks. This democratization and need for co-existence between Big Data platforms comes with new architectural requirements. For example, to exploit larger memories, the current generation of Big Data platforms [4, 31] provide external, distributed caching mechanisms such as HDFS caching [6] and Tachyon [24] to cache HDFS data in memory.

The memory-storage hierarchy in database systems is currently undergoing a radical evolution in the context of Big Data systems. Traditional relational databases take ownership of their data and store it in proprietary formats both on-disk and in dedicated buffer pools. On the other hand, SQL-on-Hadoop systems such as Impala and Hive [5, 22] store data in HDFS using open file formats (e.g, Parquet, Text), but do not provide automatic caching mechanisms. In IBM Big SQL, as we present in this paper, we take another step in this evolution by replacing the traditional buffer pool with the HDFS cache [6], an *external* cache. External caches allow us to retain the performance benefits of avoiding disk I/O, not only in Big SQL but also in other data analytics applications that share the cache with it. This solution also avoids the fragmentation of resources that occurs when different applications maintain their own specialized caches or buffer pools.

However, this design introduces new challenges. Since data in external caches is stored in the original file format, different applications must first convert it into their internal representations as needed. As a result, external caches help reduce I/O costs, but not necessarily CPU costs. Further, whereas all data access in a database system must go through the buffer pool, external caches may be used more *selectively*. Since buffer pools necessarily insert an object (page) into the cache on a cache miss, most caching algorithms, such as LRU, focus on which pages to evict from the buffer pool. However, a cache miss is handled differently in our setting. First, insertions into the external cache are costlier, as they may be asynchronously executed by a separate cache management process, competing for resources with the application that needs the data, such as the SQL system. In fact, in our experiments, we observed that traditional caching algorithms which assume that all data accesses go through the cache, often result in worse perfor-

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
 ISBN 978-1-4503-4525-5/16/10…$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987553

mance than simply bypassing the cache and reading the data directly from secondary storage. Second, since applications can bypass the cache on a cache miss, the decision of what to insert into the cache is as important as what to evict from it. Finally, since a shared cache attempts to ensure a high cache hit rate for various different data processing applications the caching algorithms must necessarily *adapt* to the workload access patterns.

In this paper, we present our experiences in addressing the aforementioned challenges using HDFS caching [6]. We show the performance benefits from using newly-developed caching algorithms that are both selective (decide what to insert) and adaptive (improve by learning the access pattern). However, we hope our experiences also motivate further discussion in the community on how to direct ongoing architectural evolution of external caching mechanisms, particularly HDFS caching. In that spirit, we highlight some of the shortcomings of these mechanisms and suggest some avenues for future development and research.

While a plethora of caching algorithms have been developed in the past, much of this paper focuses on how external caching mechanisms impose new design objectives for caching algorithms. First, these algorithms must be selective (decide what objects to insert) and must deal with great variance in object sizes. We present a new algorithm, SLRU-K, which is a variant of the classic LRU-K [28] algorithm, adapted to the external caching scenario. Second, the traditional tradeoff between caching objects based on recency and frequency of data accesses is accentuated in this scenario. To strike a better balance between recency and frequency, we propose a novel algorithm, EXD, which makes use of a single parameter that determines the weight of frequency vs. recency of data accesses. This algorithm also takes into account the cost of a cache miss, and the probability of re-access for each object. Finally, we find that performance of caching algorithms is sensitive to the choice of parameters. Since we would like to perform well on diverse and time-varying workload access patterns, any fixed choice of parameters leads to suboptimal performance. Therefore, we develop parameter-free, adaptive variants of the different algorithms that automatically tune their behavior to the observed access pattern.

Our contributions can be summarized as follows:

- We propose selective, adaptive caching algorithms (`Adaptive SLRU-K`, `Adaptive EXD`).
- We describe the architecture of IBM Big SQL and its use of the HDFS cache.
- We show that our proposed algorithms outperform existing static algorithms on diverse workloads: synthetic workloads, batch workloads (using a TPC-DS like benchmark) and a mix of concurrent batch and interactive queries.
- We discuss our experiences using HDFS caching for Big SQL and provide insights for future work.

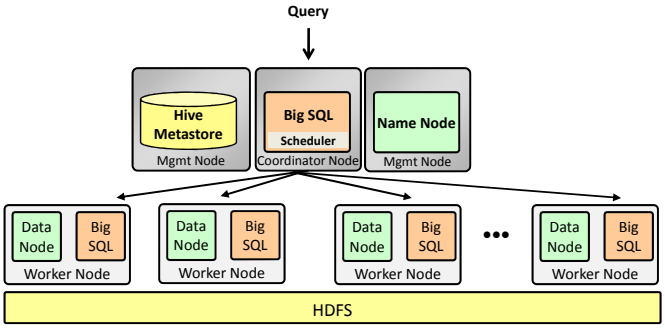

Figure 1. Big SQL Architecture

2. Overview of Big SQL

We now give brief overview of Big SQL, IBM's SQL-on-Hadoop offering, which is part of the IBM® InfoSphere® BigInsights™ data platform. A full description of the Big SQL architecture and capabilities can be found in [17]. Figure 1 presents the overall Big SQL architecture.

Big SQL leverages IBM's state-of-the-art relational database technology to execute SQL queries over HDFS data, supporting all the common Hadoop file formats; text, sequence, Parquet and ORC files. Big SQL follows the traditional shared-nothing parallel architecture. More specifically, it consists of a coordinator node and a set of worker nodes. The incoming SQL statements are compiled and optimized at the coordinator node to generate a parallel execution query plan. A runtime engine then distributes the parallel plan to worker nodes and orchestrates the consumption and return of the result set. Once a worker node receives a query plan, it dispatches special processes that know how to read and write HDFS data natively. Big SQL employs a state of the art cost-based query optimizer that exploits several statistics about the data to produce an efficient query plan.

Big SQL supports a vast range of SQL standard constructs, allowing existing database applications to be executed directly on Hadoop data. More specifically, it provides support for stored procedures, SQL-bodied functions and a rich library of scalar, table and online analytical processing (OLAP) functions among others. In this way, Big SQL creates an opportunity to reuse and share application logic among database platforms.

A fundamental component in Big SQL is the scheduler service which acts as a bridge between the Big SQL workers and HDFS. More specifically, the scheduler assigns HDFS blocks to database workers for processing on a query by query basis. It identifies where the HDFS blocks are, and decides which database workers to include in the query plan, ensuring that work is processed efficiently, as close to the data as possible. The assignment is done dynamically at run-time to accommodate failures: the scheduler uses the workers that are currently available. If a new node is added to the database cluster, it can be considered immediately by the scheduler for the new queries. Similarly, if a node crashes or the cluster is scaled down, the scheduler immediately detects

this change and chooses database workers for future queries accordingly. In case of partitioned tables, which are common in SQL-on-Hadoop environments, selection predicates are pushed down to the scheduler to eliminate partitions that are not relevant for a given query.

Big SQL operates on top of unpartitioned and partitioned tables. The tables are partitioned based on the Apache Hive [5] partitioning scheme. If a query contains predicates on the partitioning columns, then Big SQL will only access the relevant partitions, thus minimizing the total amount of data read from HDFS. Each partition consists of one or more HDFS files of different size which are all accessed when the partition is accessed. Big SQL utilizes the Hive metastore to maintain statistics such as table definitions, location, and storage format among others. This means it is not restricted to tables created and/or loaded via the Big SQL interface. As long as the data is defined in the Hive Metastore and accessible in the Hadoop cluster, Big SQL can seamlessly process it.

3. Overview of HDFS Caching

The HDFS cache [6] is an explicit caching mechanism that allows users to specify directories or files to be cached by HDFS. The HDFS namenode will communicate with datanodes that have the corresponding blocks on disk, and instruct them to cache the blocks in off-heap memory. The HDFS cache implements its own algorithms to decide which replica of a given block will be cached, and in which datanode. The namenode is also responsible for coordinating all the datanode off-heap caches in the cluster. To do so, it periodically receives heartbeats from the datanodes that describe the state of their cache.

The users can use the HDFS cache APIs to create, add, and remove HDFS cache pools. Each cache pool can host a set of HDFS directories and files[1], and has Unix-like permissions. The users can use the HDFS cache APIs to insert and remove HDFS files/directories at a specific cache pool. They also have the flexibility to choose their own cache replication factor as well as a maximum time-to-live for each cached file/directory.

Although most existing SQL-on-Hadoop solutions (e.g., Impala [1], Hive [6]) provide support for tables cached in the HDFS cache, they require the users to manually pin the tables in the HDFS cache. To the best of our knowledge, Big SQL is the first system to exploit automatic caching using the HDFS cache.

4. Big SQL Caching Framework

We now provide an overview of Big SQL's caching framework. We start by describing the Big SQL enhancements needed to allow Big SQL to cache selected table partitions into the HDFS cache. We then discuss the requirements that a caching algorithm should satisfy in this setting.

[1] The HDFS terminology is cache directives.

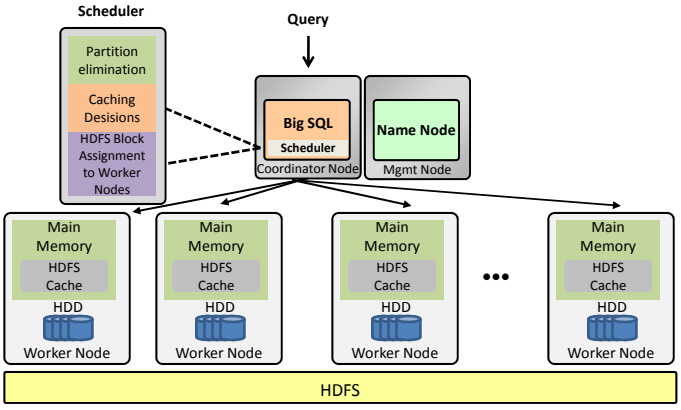

Figure 2. Caching in Big SQL

4.1 System Implementation

We now describe the implementation of a caching framework in Big SQL using the HDFS cache. Figure 2 presents an overview of the Big SQL caching framework.

As noted in Section 2, the scheduler component acts as the bridge between the SQL runtime engine and Hadoop. The scheduler maintains information about where and how data is stored on HDFS. Moreover, it is aware of which data objects are accessed for each query. For this reason, we incorporated our caching algorithms in the scheduler service.

Our caching algorithms operate at the level of table partitions, considering unpartitioned tables as consisting of a single partition. While each partition may itself consist of multiple HDFS files of different sizes, the caching algorithm maintains metadata (see Section 5) per-partition rather than per-file to minimize memory footprint. For every scan operation in a query, the Big SQL scheduler first eliminates unnecessary partitions, and then invokes the caching algorithm to decide which partitions to insert into the HDFS cache. Note that Big SQL performs I/O elimination at the partition level and thus our caching algorithms use partitions as the units for caching. The scheduler uses the appropriate HDFS APIs [6] to instruct HDFS to cache a partition. Note that the actual cache insertions are performed by HDFS and not by Big SQL.

During query execution, the Big SQL scheduler always attempts to assign data to worker nodes optimizing for data locality in a best effort fashion, giving priority to memory locality, and then disk locality. More specifically the scheduler, gathers the locations of all the replicas of a given block that will be accessed by the query, and attempts to first assign the cached replicas to the workers that host them, then assigns the local on-disk replicas, and finally incorporates accesses to remote replicas.

Data on HDFS may occasionally change. For example, deletion of files, file appends, or file additions in a table or partition can be performed without going through the Big SQL interface. For this reason, our caching algorithms

maintain a timestamp for each partition in the cache. The timestamp is the time of the latest modification of all the files that comprise the partition. When the partition is accessed again, the algorithm checks the latest modification time for this data to identify potential data changes since the last time this data was accessed. In case there has been a change, the algorithm compares the new size of the data with the size of the previous access. If the new size is smaller than the one stored in the metadata, then one or more deletion operations have been performed and some files would no longer reside in the HDFS cache. This is because, when a cached HDFS file is deleted, HDFS automatically removes it from the HDFS cache. In this case, the caching algorithm only updates its metadata (latest modification time, new data size). In case the new size is equal or greater than the one stored in the metadata, the partition is removed from the cache, and the algorithms attempt to re-insert it into the cache taking into account its new data size.

4.2 Caching Algorithm Requirements

In this section, we present the properties that a caching algorithm should have in order to be effective in the context of Big SQL. The requirements are the following:

- **Support for Online Caching**: The Big SQL worklods typically consist of ad hoc, analytic queries whose access pattern evolves over time. For this reason, we focus on online caching algorithms that unlike offline algorithms, do not assume any knowledge of the future workload. The Big SQL caching algorithm is invoked every time a table partition is accessed. Upon a cache miss, the algorithm decides whether the newly-accessed partition should be inserted in the cache, and if there is not enough free space, which cached partitions should be evicted in order to accommodate the new partition.
- **Support for Selective Cache Insertions**: Typically, caching algorithms such as `LRU-K` [28], are focused on which partitions should be evicted from the cache to accommodate a newly-accessed partition. These algorithms always insert the newly-accessed partition in the cache. However, this policy is not applicable to HDFS cache, because cache insertions are performed by an external process, which is not part of the Big SQL query engine. This process competes for resources (e.g., I/O bandwidth) with the Big SQL engine and can actually slow down the processing of the workload. In Section 6, we present experimental results that highlight this problem. For this reason, a caching algorithm for Big SQL should selectively perform insertions in order to minimize the HDFS cache insertion overheads.
- **Ability to adapt to various workload patterns**: Big SQL workloads exhibit various access patterns. For example, one application may access a particular dimension table in a star schema much more frequently than the other tables. On the other hand, another application may access the same portion of the fact table frequently for a while because the analytics works on a time window but then this time window shifts. Hence, the most-recently-accessed data items are not always the same as the most-frequently-accessed ones. Some caching algorithms, such as the `LFU` (Least Frequently Used) algorithm, base their caching decisions on the frequency of data accesses. On the other hand, algorithms such as the `LRU` (Least Recently Used), take the recency of data accesses into account. Depending on the characteristics of a particular workload, one type of algorithm can be more effective than the other. Since the Big SQL workload access patterns evolve over time, the Big SQL algorithms must be able to adjust their behavior according to the current access pattern. For this reason, we designed *adaptive* caching algorithms that decide how much weight they should give to frequency vs. recency by observing the workload performance.

5. Big SQL Caching Algorithms

In this section, we present the caching algorithms that we developped for Big SQL. We first briefly introduce a knapsack formulation of the caching problem based on which our algorithms are designed. We, then, discuss the Big SQL algorithms in more detail.

5.1 Caching Problem Foundations

We now give a high-level overview of the caching problem and we also define notation that we will later use when presenting the Big SQL caching algorithms.

The task of maximizing the expected performance of a cache has been modeled in literature as a knapsack problem [18, 19]. In this well-known formulation, it is assumed that caching an object provides certain benefit (future accesses to the object will be hits) and the cache policy has to maximize the total expected benefit from the cache given that the total size of the cached objects cannot exceed the size of the cache. Most caching algorithms can be viewed as different solutions to this knapsack problem that differentiate based on how they estimate the probability of re-accessing an object in the future.

Let the table partitions be denoted by $i = 1, \ldots, n$, denote the size of partition i by s_i and let $P_i(t)$ be the probability that the partition i will be referenced at time t. Let us denote by c_i, the benefit from the presence in cache (or the cost of a miss) of partition i. The benefit c_i may depend on s_i and possibly other characteristics of the partition including its source (hard disk, SSD, etc.) In the context of Big SQL, we assume that the cost of miss c_i of partition i is proportional to the partition's size s_i. This is a reasonable assumption since its partition consists of one or more files to read from a hard disk or over the network. Moreover, assume that each partition i has a weight which changes over time and is defined as: $W_i(t) = c_i P_i(t)$.

If the cache has a capacity C, then an optimal set $M(t)$ of partitions to be in cache at time t is one that maximizes the

total benefit of having the partitions in the cache:

$$\sum_{i \in M(t)} c_i P_i(t)$$

subject to the capacity constraint

$$\sum_{i \in M(t)} s_i \leq C \ .$$

The approximate solution for this problem is well-known and the details are ommitted in the interest of space. The solution suggests that to determine which partitions should be stored in the cache at a future time t, the caching algorithm should maintain the partitions in a sorted list according to the `ratios` $R_i(t) = \frac{c_i P_i(t)}{s_i} = \frac{W_i(t)}{s_i}, 1 \leq i \leq n$. Then, it should select partitions with the highest `ratio` $R_i(t)$ from the list, and add them in the cache until it is full. This approximate solution is the basis of our algorithms.

The knapsack solution requires knowledge of $W_i(t)$, and thus $P_i(t)$, which is the probability that the partition i will be referenced at time t. It is obvious that an online algorithm cannot know *a priori* the value of this probability for future point in time. The Big SQL algorithms estimate the probability of access based on the workload history. More specifically, at *current time* u, the algorithms statistically or heuristically estimate the probability based on their knowledge of the workload history up to time u. Let's denote this probability as $p_i(u)$. Our algorithms make the assumption that $P_i(t) \simeq p_i(u)$. Thus, we can also assume that $W_i(t) \simeq w_i(u) = c_i p_i(u)$ and that $R_i(t) \simeq r_i(u) = w_i(u)/s_i$. As we will show in the following section, different algorithms use different probability estimation formulas.

Moreover, in order to make fast caching decisions, the Big SQL caching algorithms assume that the probability function $p_i(u)$ has the following property:

ASSUMPTION 5.1. *If $p_i(u) > p_j(u)$ at a time u then $p_i(u+\Delta u) > p_j(u+\Delta u)$ for all partitions i, j that have not been accessed during the interval $(u, u+\Delta u]$. Thus, if $r_i(u) > r_j(u)$, then $r_i(u+\Delta u) > r_j(u+\Delta u)$.*

Consider a sorted list that contains information about the partitions residing in the cache at time u. The partitions in the list are sorted in ascending order of the `ratio` $r_i(u)$. Let's assume that we want to maintain the list sorted as partitions are accessed over time and their probabilities of re-access change. The next partition access happens at time $u+\Delta u$. According to Assumption 5.1, the relative order of those partitions in the list that were not accessed during the time interval $(u, u+\Delta u]$, does not need to change. Only the position of the currently-accessed partition needs to be updated. In this way, we can avoid re-sorting the whole list after each partition access.

5.2 Caching Algorithm Template

In this section, we provide a template algorithm that is invoked each time a partition is accessed. Our caching algorithms specialize this template by providing their own definitions of $p_i(u)$, and thus $w_i(u)$ and $r_i(u)$. We present the pseudocode in Algorithm 1. The algorithm uses a global integer counter `Time` to simulate time which is incremented each time a partition is accessed.

Algorithm 1: Caching Algorithm Template

```
Data: Partition b of size s_b, Used, Capacity, CacheState,
      History
Result: true if b is inserted in the cache, false otherwise
1  Time++;
2  Create or retrieve info about b in History;
3  Set last access time of b to Time;
   // Handle Cache Hit
4  if Partition b is in the cache then
5      Set b's ratio to r_b(Time) in the CacheState;
6      return false;
   // Handle Cache Miss when b fits in cache
7  if s_b + Used ≤ Capacity then
8      Insert b in the CacheState with ratio r_b(Time) ;
9      Used=Used+s_b;
10     Insert b into the cache;
11     return true;
   // Handle Cache Miss when b does not fit in
      cache
   // Evaluate whether b should be inserted in the
      cache using the weight heuristic.
12 Compute the weight w_b(Time) of b;
13 Set total weight of partitions to be evicted sumWeights = 0;
14 Set freeSpace = Capacity - Used;
15 foreach partition next in CacheState in ascending order
     of ratios do
16     if sumWeights + w_next(Time) < w_b(Time) then
17         sumWeights = sumWeights + w_next(Time);
18         freeSpace = freeSpace + s_next;
19         Add next to the Eviction List;
20         if freeSpace ≥ s_b then
21             exit the loop;
22 if freeSpace < s_b then
23     return false;
24 Evict from the cache all the partitions in Eviction List;
25 Insert b into the cache and CacheState with ratio r_b(Time);
26 return true;
```

The algorithm maintains two data structures: the `CacheState` and the `History`. The `CacheState` contains all the information about the partitions that are currently in the cache, including the `ratio` $r_i(u)$ at time u and their size. The `CacheState` is implemented as a list sorted by $r_i(u)$ in ascending order. In practice, by making use of a probability function that satisfies Assumption 5.1, a caching algorithm can maintain the correct sorted order as partitions are ac-

cessed, without updating the ratios of all the partitions in the cache each time.

The History contains metadata about all the partitions that have been accessed in the past, such as their size, and time of last access, and can be implemented as a hash table keyed by the partitions. Since the History grows over time, one can restrict the number of entries in this data structure, or remove from History partitions that have not been accessed for a long period of time.

Let us consider a cache of size Capacity. Let Used be the current size of the cache used to store partitions. When a partition b is accessed, the Time counter is incremented by 1, and if the partition is contained in History then the latest metadata about the partition is retrieved. If the partition b is not present in History then a new entry is created for it (Lines 1-3).

The algorithm then checks whether the partition is already in the cache (*cache hit*) or not (*cache miss*). In case the partition b is already in the cache, the algorithm needs to update the partition's corresponding metadata, namely, its latest access time as well as its ratio $r_b(\texttt{Time})$. Note that since the CacheState is implemented as a list sorted by the ratios of the cached partitions, we need to remove partition b from the list, update its ratio, and then re-insert it to keep the correct sorted order (Lines 4-6). We would like to emphasize that if the probability function of the algorithm satisfies Assumption 5.1, then we do not need to update the ratios of the cached non-accessed partitions to reflect the new value of the Time counter since the sort order is correctly maintained.

If the partition is not contained in the cache (*cache miss)*, then the algorithm checks whether there is enough free space in the cache to accommodate the partition. If so, the partition is inserted into the cache (Lines 8-12). Otherwise, the algorithm uses the weight heuristic to identify whether the partition should be cached.

The weight heuristic attempts to minimize insertions in the cache, since they can negatively affect the workload performance. The heuristic applies a greedy approach to maximize the *total weight* of the cache each time a cache insertion decision needs to be made. Following the approximate knapsack solution, the heuristic traverses the partitions stored in CacheState in *ascending* order of ratios, attempting to identify candidates for eviction in order to accommodate partition b. The heuristic maintains a list of candidate partitions for eviction, namely Eviction List. At every step, the algorithm checks whether by adding the partition currently under consideration to the Eviction List, the total weight of the candidate partitions for eviction would be less than the weight of the newly-accessed partition b. In this case, the partition currently under consideration is added to the Eviction List (Lines 18-23). Otherwise, the partition currently under consideration is not added to the Eviction List, and the algorithm proceeds with the next partition in the sorted list. The heuristic terminates if enough space for the newly-accessed partition is found (Lines 22-23), or if all the partitions in the list have been examined. If the total size of the partitions in the Eviction List is enough, then partition b is inserted in the cache (Lines 24-30).

5.3 Estimating the Probability of Access

We now present in detail the SLRU-K and EXD algorithms. Both algorithms follow the template presented previously but utilize different definitions of $p_i(u)$. Because of the different nature of the probability functions, the two algorithms maintain different types of metadata per partition. More specifically, the EXD algorithm requires fewer metadata items per partition than the SLRU-K algorithm.

5.3.1 The SLRU-K algorithm

The Selective LRU-K (SLRU-K) algorithm is an extension of the LRU-K algorithm that takes into account the variable size of the partitions. As opposed to LRU-K, the SLRU-K algorithm does not insert each accessed partition into the cache, but rather selectively places partitions in the cache using the weight heuristic.

For each partition i, the SLRU-K algorithm maintains a list $L_i = [u_{i1}, ..., u_{iK}]$ of the K most recent accesses sorted in descending order. Thus, the time of the last access of the partition is represented by u_{i1} and the time of the Kth most recent access is represented by u_{iK}. This list is updated when the partition is accessed, by introducing a new value (time of last access) in the head of the list and dropping the last value, if needed, in order to keep the list limited to at most K values.

For a given partition i and current *time* u, let $T_i(u) = u - u_{iK} + 1$ be the number of partition accesses since partition i's Kth most recent access. The SLRU-K algorithm estimates the probability that partition i will be accessed at time $u+1$ as

$$p_i(u) = \frac{K}{T_i(u)} \tag{1}$$

where $T_i(u)$ is the total number of accesses in the interval (see above) that includes the K most recent accesses of partition i until time u. This probability is estimated statistically and the proof is ommitted in the interest of space. Note that the estimate $p_i(u)$ is changing over time as more accesses are happening, and the value of $T_i(u)$ changes. The SLRU-K algorithm takes into account the new values of these estimates since the list of the last K accesses of each partition is updated. Finally, it can be shown that the probability function of the SLRU-K method has the property described in Assumption 5.1. The details are ommitted in the interest of space.

5.3.2 The EXD algorithm

We now present the Exponential-Decay (EXD) caching algorithm. The algorithm implements the template presented in Section 5.2, and makes use of a single parameter (a) that determines the weight of frequency vs. recency of data

accesses. In this section, we focus on how the EXD algorithm approximates the probability $p_i(u)$.

DEFINITION 5.1. Denote by u_{ij} the j^{th} most recent access time of partition i. For a constant parameter $a > 0$ define the score $S_i(u)$ of partition i at current time u as $S_i(u) = e^{-a(u-u_{i1})} + e^{-a(u-u_{i2})} + \cdots$

As shown, the score of a partition depends on the value of the parameter a. The value of this parameter essentially determines how recency and frequency are combined into a single score. The larger the value of a, the more emphasis on recency versus frequency. The value of a can also be chosen adaptively as we will describe in Section 5.4.

The EXD algorithm assumes that for a given partition i, at the current time u, the score $S_i(u)$ is proportional to $p_i(u)$. Notice that the algorithm does not require exact knowledge of the values of $p_i(u)$ of the accessed partitions. It rather needs to know the relative order of the ratios $r_i(u)$ of all different partitions. For this reason, the algorithm substitutes the partition's probability function $p_i(u)$ with the partition's score $S_i(u)$ in Algorithm 1.

It follows that at any given point in time u, the EXD algorithm needs to compute the score $S_i(u)$ of the partitions. The following proposition describes how we can efficiently compute the score of a partition at a specific point in time, given only the time of its last access, and the corresponding score at that time. Note that, unlike the SLRU-K algorithm which needs to maintain the last K access times for each partition, the EXD algorithm reduces the memory footprint by keeping only the time of the last access of each partition.

DEFINITION 5.2. *For a partition i, the score $S_i(u_{i1} + \Delta u)$ can be calculated if we only keep the most recent time of access u_{i1} and the score $S_i(u_{i1})$.*

Proof. Obviously, if partition i is not accessed during the interval $(u_{i1}, u_{i1} + \Delta u]$, then

$$S_i(u_{i1} + \Delta u) = S_i(u_{i1}) \cdot e^{-a\Delta u} \tag{2}$$

and if it is accessed at time $u_{i1} + \Delta u$ for the first time after time u_{i1}, then

$$S_i(u_{i1} + \Delta u) = S_i(u_{i1}) \cdot e^{-a\Delta u} + 1 \ . \tag{3}$$

□

It follows that the score $S_i(u)$ can be calculated for any time $u > u_{i1}$ before the next partition access. Furthermore, the scores decay exponentially and can be approximated by zero after they drop below a certain threshold. This allows us to stop maintaining history for partitions that have not been accessed for a long time. Finally, the scoring function (and thus the probability function) of the EXD method has the property described in Assumption 5.1.

Algorithm 2: Adaptor

Data: boolean CacheHit, boolean PartitionInserted, long partitionSize
Result: new value of algorithmic parameter newParameter

```
1  eventNo++;
2  Update the BHR(currentParameter) and
     BIR(currentParameter) based on the values of CacheHit,
     PartitionInserted, and partitionSize;
3  if (eventNo == maxEventsPerRound) then
       // end of current round
4      eventNo = 0;
       // Update the BHR and BIR values taking
          into account all the rounds so far
5      BHR(currentParameter) =
         weightedAverage(previousBHR(currentParameter),
         BHR(currentParameter));
6      BIR(currentParameter) =
         weightedAverage(previousBIR(currentParameter)),
         BIR(currentParameter));
       // Select the new value of the parameter
7      Group the parameters in CandidateValues according to
         their corresponding BHR observed so far;
8      if (no time for exploration) then
9          selectedGroup = pick group with highest
             representative BHR;
10     else
11         selectedGroup = pick group with probability
             proportional to its BHR;
12     newParameter = pick the parameter value in
         selectedGroup with the minimum BIR value;
13     return newParameter to the caching algorithm;
14 else
       // not the end of current round
15     newParameter = current value of the parameter;
16     return newParameter to the caching algorithm;
```

5.4 Adaptive SLRU-K and EXD

Both the EXD and the SLRU-K algorithms are parameterized. The behavior of the algorithms can significantly change based on the values of a and K. As we will show in Section 6, there is no single value of a (or K) that works well across all possible workloads.

Figuring out the best value of the algorithmic parameter is difficult for two reasons: (1) The optimal value of the parameter depends on the workload access pattern, and (2) The workload access pattern is not stable over time. In this section, we present an adaptive algorithm (Adaptor) that automatically adjusts the value of the algorithmic parameter in order to improve overall performance.

The Adaptor can be used with both the SLRU-K and the EXD methods. It operates along with the caching algorithm, in a separate thread, and exchanges information with it. Each partition access is treated as an event. At every event, the

caching algorithm informs the Adaptor whether the event was a cache miss or a cache hit, and whether the partition was inserted into the cache. The Adaptor uses this information to adjust the algorithmic parameters over time.

The Adaptor takes into account two metrics when making decisions about the value of the algorithmic parameter. The *primary metric* is the *byte hit ratio (BHR)* which is a standard comparative performance metric used in prior work on caching variable-size partitions [8, 11, 29, 30]. The *BHR* is the fraction of the requested bytes that was served from the cache. The higher the *BHR*, the fewer I/O requests need to be made, and the greater the overall performance. As in previous work, our primary goal is to maximize the BHR. In an external caching system, such as HDFS cache, cache insertions compete for resources with the process that needs to access the data, and thereby slow down the workload. To quantify the overhead of each algorithm with respect to cache insertions, we introduce a *secondary metric*, namely the *byte insertion ratio (BIR)*. The *BIR* is the fraction of the requested bytes that the caching algorithm decided to insert into the cache.

In our setting, it is desirable to maximize the *BHR* so that the hot set is always cached while maintaining a low *BIR* if possible. Our Adaptor constantly evaluates the behavior of the caching algorithm by measuring these metrics, and its *primary goal* is to maximize the *BHR*. From all the values of the algorithmic parameter that maximize the *BHR*, the Adaptor prefers the one that minimizes the *BIR*, since it reduces the cost of insertions in the cache.

The pseudocode for the Adaptor is presented in Algorithm 2. The algorithm uses a set of pre-defined parameter values, namely *CandidateValues*. In case of the SLRU-K algorithm, the *CandidateValues* set contains the following values for the *K* parameter: $1, 2, 4, 6, 8$. In the case of the EXD algorithm, the *CandidateValues* set contains six *a* values equally-spaced in the log space with $a_{min} = 10^{-12}$ and $a_{max} = 0.3$. These values cover a large range of potential parameter instantiations that can successfully be applied in many workload scenarios. For each potential value of the algorithmic parameter $i \in$ *CandidateValues*, the Adaptor maintains the observed *BHR*(i) and *BIR*(i) achieved with the value i so far. Initially, the parameter is randomly assigned a value from the set of possible values.

Caching Algorithm	Selective Insertions	Adaptiveness to the Access Pattern
LRU-K	No	No
LFU	No	No
GDS	No	No
SLRU-K	Yes	No
EXD	Yes	No
Adaptive SLRU-K	Yes	Yes
Adaptive EXD	Yes	Yes

Table 1: Comparison of various online caching algorithms

The algorithm operates on rounds that consist of a fixed number of events. After every event, the Adaptor updates the *BHR* and *BIR* values observed for the current value of the parameter (*currentParameter*), based on the information received from the caching algorithm (Line 2).

When the last event of the round is processed, the *BHR* and the *BIR* values that correspond to the current parameter value are updated using a weighted average over the observed *BHR* and *BIR* values across all rounds, giving more emphasis on the observations of the last round (Lines 3-6). The Adaptor then re-evaluates the value of the algorithmic parameter. The re-evaluation process consists of three steps. In the first step, the Adaptor groups the parameter values of the *CandidateValues* set, according to their observed *BHR* so far. Parameter values with *BHR* values within a certain threshold of each other are placed in the same group (Line 7). Each group has a representative *BHR* value, which is the average of the *BHR* of its members. In the next step, the Adaptor picks the group with the highest representative *BHR* (Lines 8,9). Occasionally, at this step, the Adaptor selects a group with probability proportional to the *BHR* of the group (Lines 10,11). This happens so that the parameter space is explored by observing the behavior of the caching algorithm for different values of the parameter. After a group has been selected, the Adaptor selects a member of this group by taking into account the *BIR* values that have been achieved so far by the members of the group. More specifically, it picks the parameter value that has resulted in the lowest *BIR* so far (Line 12).

After the value of the parameter has been selected, the Adaptor informs the caching algorithm of the new value (Lines 13,16). The caching algorithm, then, updates the ratios of the partitions in the History and the CacheState to reflect the new value.

Table 1 compares our adaptive algorithms with various well-known caching algorithms with respect to the properties described in Section 4.2. Note that the table presents only online caching algorithms and compares them based on their support for selective cache insertions, and adaptiveness to various workload access patterns. The GDS algorithm presented in the table, is developed for web caching. It is a parameter-free algorithm that is able to accommodate various file sizes and has been shown to outperform various algorithms for web caches [11]. As shown in the table, only the Adaptive EXD and Adaptive SLRU-K algorithms satisfy all our requirements. An experimental evaluation of these algorithms is presented in the following section.

6. Experimental Evaluation

We now provide an experimental evaluation of our proposed algorithms with state-of-the-art caching policies.

6.1 Experimental Setting

For our experiments, we use a cluster of 10 nodes. One of the nodes hosts the HDFS NameNode, the Big SQL coordi-

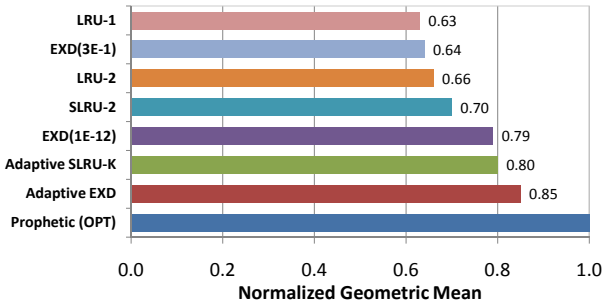

Figure 3. Comparison of various caching algorithms using the TPC-DS like workload

nator, the scheduler, and the Hive Metastore. The remaining 9 nodes are designated as "compute" nodes. Every node in the cluster has 2x Intel Xeon CPUs @ 2.20GHz, with 6x physical cores each (12 physical cores total), 8x SATA disks (2TB, 7k RPM), 1x 10 Gigabit Ethernet card, and 96GB of RAM. Out of the eight disks, seven are used for storing HDFS data. Each node runs 64-bit Red Hat Enterprise Linux Server 6.5. We use the implementation of the caching framework described in Section 4, using InfoSphere BigInsights 3.0.1 enterprise release, and test end-to-end system performance. In all our experiments, we intentionally avoided using large caches so that we can stress the caching algorithms.

6.1.1 TPC-DS Like Workload

We now present cluster experiments using a workload inspired by the TPC-DS benchmark[2]. This workload is published by Impala developers[3], and has previously been used to compare the performance of various SQL-on-Hadoop systems (e.g., [2], [16]). The workload consists of 20 queries that include multi-way joins, aggregations, and nested subqueries. The fact table is partitioned, whereas the small dimension tables are not partitioned. We use a 3TB TPC-DS database, and a 300GB HDFS cache.

We compare the different caching algorithms with a theoretically optimal reference algorithm, which we call the `Prophetic prefetcher`. Before running each query, this algorithm uses prior knowledge of the entire workload trace to prefetch as much of the data accessed by the next query as fits in the cache. As a result, all but 2 of the 20 queries ran entirely in memory. Further, the evaluation of `Prophetic prefetcher` only measures the execution time of the queries, ignoring the time to prefetch the data into memory[4]. We also compare with the well-known `LRU-K` method. The `LRU-K` algorithm extended to accommodate variable-size objects has been evaluated in the context of web caching [11] only when $K = 1$. We further evaluate the extended `LRU-K` algorithm for multiple values of K. We note that, the main difference between the `LRU-K` and the `SLRU-K` algorithms is that the former inserts every accessed partition into the cache whereas the latter performs selective cache insertions. the performance of the `GDS` algorithm is similar to that of the `LRU` algorithm and is ommitted. For each algorithm, we performed the experiment 3 times using a warm HDFS cache, and report the average over the 3 runs.

Figure 3 shows the geometric mean of the query runtimes for various caching algorithms relative to the query runtimes produced by the offline `Prophetic Prefetcher`. As shown, the adaptive algorithms achieve the best performance. The `Prophetic prefetcher` was only about 15% faster than the `Adaptive EXD` algorithm even though it had *a priori* knowledge of the entire workload. The remaining algorithms were not as efficient as the adaptive algorithms. For example, the `LRU-1` algorithm achieved 63% of the `Prophetic Prefetcher's` performance.

Figure 4 shows the runtime of each query relative to the the runtime produced by the `Prophetic Prefetcher`. Ideally, a caching algorithm should produce query runtimes close to the ones produced by the `Prophetic Prefetcher`. As shown in the figure, the adaptive algorithms generally resulted in query runtimes close to those observed when the `Prophetic Prefetcher` was used. The `LRU-1` algorithm, on the other hand, did not perform as well as the adaptive methods. When comparing the best performing online algorithm (`Adaptive EXD`) with the `LRU-1` algorithm, we observe that all but one of the queries experienced speedups ranging from $1.03X$ to $2.3X$, and the geometric mean of the speedups was $1.34X$.

Finally, if we consider the workload's total elapsed time, this was 2713 seconds when using the `LRU-1` method and 2556 seconds with the `LRU-2` method. The total elapsed time using the `Adaptive EXD` algorithm was 1711 seconds. This is an important difference, especially if we consider that the best possible performance that can be achieved by an offline algorithm is 1544 seconds (`Prophetic Prefetcher`).

We also performed experiments with other values of the parameter K. The behavior was similar to the `LRU-2` and `SLRU-2` methods and these results are omitted in the interest of space. Our results show that: (1) the adaptive algorithms gracefully adapt over time to produce the best performance results, and (2) the performance achieved is close to the one achieved by a hypothetical offline algorithm that prefetches the data needed by each query.

6.2 Hotset experiment

The goal of this experiment is to show which algorithms can correctly identify the workload's *hotset*, and how performance is affected. Our evaluation compares the various caching algorithms with the `HotSet Prefetcher`, an algorithm that has *a priori* knowledge of the entire workload, prefetches and caches the *hotset* of partitions.

The TPC-DS like queries that we used in the previous experiment access a wide range of data that keeps evolving over time making it difficult to identify the workload's

[2] http://www.tpc.org/tpcds/

[3] https://github.com/cloudera/impala-tpcds-kit

[4] Recall that reading the data into the cache incurs additional cost that needs to be paid by the HDFS cache

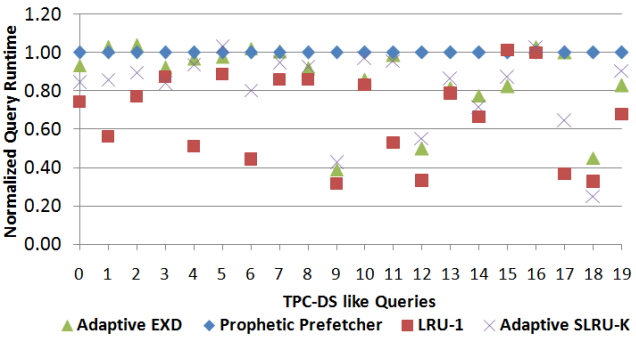

Figure 4. Normalized Query Runtime for the TPC-DS like workload

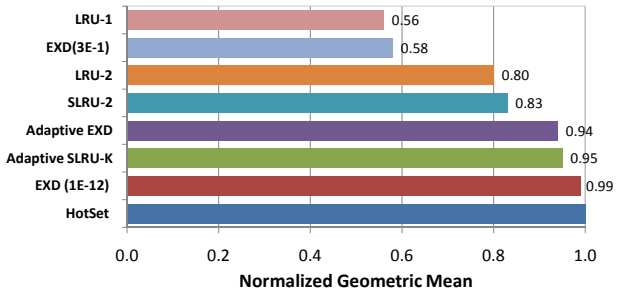

Figure 5. Comparison of various caching algorithms using the synthetic workload

hotset, and use the `HotSet Prefetcher` to upper-bound the performance. [5] For this reason, we created a workload that operates on the 1TB `store_sales` TPC-DS fact table, and has a clear hotset. In this way, we can evaluate which caching algorithms are able to identify this hotset.

Our workload consists of 50 queries that contain selections, projections and aggregations. We have observed that corporate users of Big SQL tend to frequently access their recent data, and more rarely their older/historical data, while creating summaries for reports. Thus, the workload's hotset consists of the 250 most recently created partitions. Each query in our workload accesses a subset of the table's partitions. A partition is accessed either from the most recent 250 partitions uniformly at random with probability 0.5 (*hotset*), or uniformly from the set of the remaining 1550 older partitions (*coldset*). The total size of the 250 most frequently accessed partitions is approximately 170GB. We used a 170GB HDFS cache so that the *hotset* fits entirely in the cache.

Figure 5 shows the performance of the algorithms that we tested. The chart plots the geometric mean of the query runtimes for each algorithm relative to the runtimes produced by the `HotSet Prefetcher`. As shown in the figure, the `EXD`(10^{-12}) algorithm provided almost the same performance as the `HotSet Prefetcher`. This is expected as this workload is essentially the best use-case for this algorithm, which gives emphasis on the frequency of the data accesses as presented in our simulation study. However, other values of a produce different (worse) performance (e.g, `EXD`(0.3)). The parameter-free, adaptive methods were able to achieve about 95% of the performance of the `HotSet Prefetcher`. When comparing the `Adaptive EXD` algorithm with the `LRU-1` algorithm, we observe that all but seven of the individual queries experienced speedups ranging from 1.08X to 6.02X, and the geometric mean of the speedups was 1.44X. This result highlights the need for adaptive caching algorithms.

The total elapsed time of the workload with the `Adaptive EXD` method was about 615 seconds, while the total elapsed time with the offline `HotSet Prefetcher` was 549 seconds.

[5] This is the reason we use the the per-query `Prophetic Prefetcher` to upper-bound performance of the TPC-DS like workload.

Note that the adaptive algorithms occasionally re-evaluate the parameter space, and thus, pay some exploration cost. Nevertheless, they are able to perform very well under various workload patterns.

Another interesting point is the `LRU-1` and `EXD`(0.3) algorithms resulted in higher total elapsed time for this workload (934 seconds and 885 seconds respectively) than a system that does not use the HDFS cache at all (837 seconds). The reason is that these algorithms perform multiple cache insertions that compete for resources with the query engine, essentially slowing down the workload. Setting an algorithmic parameter incorrectly can result in unexpected system behavior.

6.3 Concurrent Workload

In this experiment, we evaluate our algorithms using a complex workload with a diverse mix of concurrent **batch** and **interactive** queries. Our goal is to investigate how the performance of interactive workloads that have low response time requirements gets affected by long running analytics workloads, such as batch queries used for reporting, running concurrently for various caching algorithms. In particular, we run batch analytics queries (the TPC-DS like workload described in Section 6.1.1) concurrently with parallel streams of interactive queries. The interactive queries are continuously executed using three parallel streams until the TPC-DS like workload finishes. We, then, evaluate how the average response time of the interactive queries gets affected by the batch queries and how the total elapsed time of the TPC-DS like workload varies with the caching method.

The interactive queries are aggregations over a single partition of a large, 1TB table. The table is a copy of the TPC-DS fact table used in the previous experiments (Section 6.2). We created a separate table for the interactive queries in order to force the batch and interactive queries to access different data sets, and thus compete more aggressively for the cache space. We used the same access pattern for the partitions of the table as in the previous experiment. More specifically, the interactive queries access a partition either from the most recent 250 partitions uniformly at random with probability 0.5, or uniformly from the set of the 1550 older par-

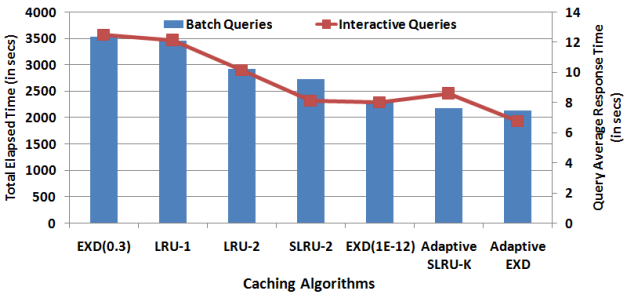

Figure 6. Comparison of various caching algorithms using the concurrent workload

titions. Our total database size is $4TB$ and our HDFS cache size is $470GB$.

To evaluate our results, we collect performance metrics for both the batch queries and the interactive queries. Figure 6 shows the total elapsed time in seconds for the TPC-DS like workload (left y-axis) as well as the average response time in seconds of the interactive queries across the three concurrent streams (right y-axis) for different caching algorithms. As shown in the figure, the adaptive, parameter-free algorithms resulted in the lowest elapsed time for the TPC-DS like workload. The TPC-DS like workload ran for 3468 seconds with `LRU-1` algorithm, and it completed in just 2145 seconds with the `Adaptive EXD` algorithm ($1.6X$ speedup). In fact, all but two of the individual queries experienced speedups ranging from $1.06X$ to $2.21X$, and the geometric mean of the speedups was $1.47X$. Moreover, it is remarkable that the higher performance for the TPC-DS workload did not come at a cost of performance for the interactive queries. On the contrary, while the interactive queries ran for an average of 12.15 seconds using the `LRU-1` algorithm, they ran in about 6.8 seconds using the `Adaptive EXD` algorithm, an effective performance gain of $1.78X$. A similar trend was also observed for the `Adaptive SLRU-K` algorithm.

Our results show that the parameter-free, adaptive algorithms, especially the `Adaptive EXD` algorithm, can provide the best performance for both the batch queries and the interactive queries.

6.4 Simulation Study

While the proposed algorithms do improve the performance of Big SQL, the performance gain does not match that from traditional buffer pools in relational databases. Simulation studies have been used in prior work [11, 20, 21, 23, 26, 28, 29, 32] to isolate and compare the performance of the algorithms without being clouded by incidental system implementation or hardware details such as CPU efficiency, I/O and network bandwidth. Based on a detailed simulation study [15] on various cache sizes, we concluded that the the proposed algorithms are nearly optimal, and the limited performance gain we observed must be attributed to other factors. More specifically, we observed that the basic `SLRU-K` and `EXD` algorithms achieve high *BHR* and low *BIR* for different workloads, but none of them individually performs well on all of them. However, the adaptive algorithms, especially the `Adaptive EXD` algorithm, achieve the best balance between *BHR* and *BIR*, effectively producing the lowest *BIR* without negatively affecting the *BHR*. Finally, none of the traditional algorithms can consistently outperform `Adaptive EXD` across different workload patterns.

7. Perspectives

Our comprehensive analysis revealed two major performance bottlenecks related to the design and implementation of the HDFS cache. First, the HDFS process responsible for caching the requested HDFS blocks in the off-heap caches of the specified datanodes is significantly slow. More specifically, the process utilizes only 30MB/sec of the available disk bandwidth per compute node. This behavior significantly affects workloads whose hotset depends on the recency of data accesses. As we briefly discussed in our simulation study, such workloads produce a large number of cache insertions because of their evolving hotset. In such cases, Big SQL cannot benefit from HDFS caching at all. This is because due to the slow cache insertions there is a high chance that the recently requested data would not reside in the HDFS cache and have to be fetched from secondary storage. For these workloads, performing selective cache insertions cannot solve the problem as the insertions must happen in order to keep up with the evolving hotset. On the other hand, the slow cache insertions do not affect workloads whose hotset depends on the frequency of data accesses. Although, initially these workloads may slow down due to the external caching process, once the HDFS cache is warm, our weight heuristic will minimize the number of cache insertions, and the overall performance will improve. The second performance bottleneck is related to the high deserialization and decompression cost while reading HDFS data. Note that the HDFS cache hosts data in the on-disk data format (e.g., Text, Parquet). As a result, the data must be deserialized and decompressed before consumed by the database workers. The deserialization process creates additional CPU overheads that can negatively affect the overall performance.

Another approach to exploit the large available memory of typical clusters in the context of SQL-on-Hadoop systems, is to implement a traditional buffer pool. Buffer pools have different characteristics than external caches. First, buffer pools store the data in the internal format of the database, and thus, avoid the extra overheads of deserialization and decompression. Second, in a database system all data accesses are typically carried out through the buffer pool. Hence, if a page (or an object) is not in the buffer pool, it is first brought there. Unlike external caches, this design avoids interference between the database workers that process the data and the process that performs the cache insertions. However, despite these benefits, buffer pools do not fit well with the Big Data platforms that contain many

frameworks, not just SQL engines. This is because unlike external caches, they do not allow data sharing across different frameworks, and they tend to fragment resources in environments where multiple processing frameworks operate on the same cluster.

Given the co-existence of many data processing frameworks on the same cluster, we believe that external caching mechanisms can provide a significant performance improvement across multiple applications, and at the same time can avoid resource fragmentation. However, our analysis demonstrates that these external caches must be able to accommodate deserialized data stored in efficient main-memory formats (e.g, columnar formats), and must also provide efficient cache insertion mechanisms. Designing, and implementing such external caches, and integrating them with various data processing frameworks can have a significant impact in the next generation Big Data processing stack. We believe that the combination of two emerging technologies, Apache Arrow [3] and Tachyon [24], can provide a solution to the problems we observed with HDFS caching and external caches in general. Arrow is an open-source initiative that provides an in-memory columnar data layout that can be shared by many processing frameworks, without deserialization. Tachyon is an in-memory file system that can be shared by all the frameworks running in the cluster. The main challenge for this combination is finding efficient caching algorithms that can adapt to different workloads, and can support the multi-tenancy inherent in the system. We believe our adaptive algorithms provide a promising solution for the former problem, and we plan to extend this work to take multi-tenancy into account. Another interesting avenue for future work is to develop caching algorithms that can exploit deeper storage hierarchies that include not only memory and HDD disks, but also non-volatile memory (NVRAM) and SDDs.

8. Related Work

There is a lot of work in cache replacement policies developed in various contexts. For brevity, we point the reader to [11, 26] for a more comprehensive survey of the existing literature. Instead, we highlight the most closely related work to place our current work in the proper context. In the context of relational databases and storage systems, there is extensive work on page replacement policies such as the LRU-K [28], DBMIN [13], ARC [26], LIRS [20], LRFU [23], MQ [32] and 2Q [21] policies. There is also recent work on SLA-aware buffer pool algorithms for multi-tenant settings [27]. Unlike our proposed algorithms, these policies operate on fixed size pages since they mainly target traditional buffer pool settings. Moreover, these policies assume that every accessed page has to be inserted into the buffer pool, thus selective cache insertions lie beyond their remit. We also note that our algorithms focus on caching raw data, unlike approaches like semantic caching [14].

Many caching policies have been developed for web caches that operate on variable size objects. The most well-known algorithms in the space are the SIZE [8], LRU-Threshold [7], Log(Size) + LRU [7], Hyper-G [8], Lowest-Latency-First [30], Greedy-Dual-Size [11], Pitkow/Recker [8], Hybrid [30], PSS [9] and Lowest Relative Value (LRV) [29]. The work in [11] has extensively compared various web caching algorithms, and has shown that the GDS algorithm outperforms them. In our experiments, we found that unlike our adaptive algorithms, GDS is not able to adjust its behavior to various access patterns.

Self-tuning and self-managing database systems have been studied in various contexts [12, 25]. In the context of caching, the ARC method [26] adapts its behavior based on the data access pattern. Unlike our algorithms, ARC operates only on fixed size objects and its adaptive design strongly depends on this assumption.

Exponential functions have been used before to model different types of behavior. For example, the work in [10] uses a power law with an exponential cuttoff to model consumer behavior. Our `Adaptive EXD` algorithm makes use of a parameterized exponential function to predict object reaccesses but adapts the function based on the workload access pattern. To the best of our knowledge, this is the first time that a caching algorithm makes use of an adaptive exponential function.

In the context of Hadoop systems, Cloudera [1] and Hortonworks [6], two major Hadoop distribution vendors allow the users to manually pin HDFS files, partitions or tables in the HDFS cache in order to speedup their workloads. The Impala [22] developers claim that the usage of HDFS cache can provide a $3X$ speedup on SQL-on-Hadoop workloads [1]. In the Spark ecosystem [31], Spark RDDs can be cached in Tachyon [24], a distributed in-memory file system. To the best of our knowledge, these systems do not use automatic algorithms but rather rely on the user to manually cache the data.

9. Conclusions

In this work we propose online, adaptive algorithms in the context of Big SQL. We experimentally show that our methods are able to adjust to various workload patterns, and outperform a variety of existing static algorithms. Our experimental results show that it is essential to use an adaptive algorithm that can automatically adjust its behavior based on the workload characteristics. This is because it is almost impossible to know the global system workload a priori, to identify the hotset over time, to pick the correct algorithm, and its corresponding parameter value. Finally, we also discuss our experiences in using external caches to improve SQL-on-Hadoop performance and we provide insights for future research and development in the context of caching in Big Data systems.

References

[1] HDFS Read Caching in Impala. http://blog.cloudera.com/blog/2014/08/new-in-cdh-5-1-hdfs-read-caching/, 2014. Accessed: 08.25.2016.

[2] TPC-DS like Workload on Impala. http://blog.cloudera.com/blog/2014/09/new-benchmarks-for-sql-on-hadoop-impala-1-4-widens-the-performance-gap/, 2014. Accessed: 08.25.2016.

[3] Apache Arrow. https://arrow.apache.org/, 2016. Accessed: 08.25.2016.

[4] Hadoop 2.0. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2016. Accessed: 08.25.2016.

[5] Apache Hive. https://hive.apache.org/, 2016. Accessed: 08.25.2016.

[6] Hortonworks: Centralized Cache Management in HDFS. https://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.3.2/bk_hdfs_admin_tools/content/ch03.html, 2016. Accessed: 08.25.2016.

[7] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching Proxies: Limitations and Potentials. Technical report, 1995.

[8] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams. Removal Policies in Network Caches for World-Wide Web Documents. *SIGCOMM Comput. Commun. Rev.*, 26(4), 1996.

[9] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE Trans. on Knowl. and Data Eng.*, 11(1), 1999.

[10] A. Anderson, R. Kumar, A. Tomkins, and S. Vassilvitskii. The dynamics of repeat consumption. WWW '14, 2014.

[11] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX*, 1997.

[12] S. Chaudhuri and V. Narasayya. Self-Tuning Database Systems: A Decade of Progress. In *VLDB*, 2007.

[13] H.-T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*, 1985.

[14] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. VLDB, 1996.

[15] A. Floratou, N. Megiddo, N. Potti, F. Özcan, U. Kale, and J.-S. Hermes. Technical Report: Adaptive Caching Algorithms for Big Data Systems. http://domino.research.ibm.com/library/cyberdig.nsf/papers/B7CCB65324B57D7E85257ED700505AAC/$File/RJ10531.pdf.

[16] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures. *PVLDB*, 7(12), 2014.

[17] S. Gray, F. Özcan, H. Pereyra, B. van der Linden, and A. Zubiri. Big SQL 3.0: SQL-on-Hadoop without compromise. http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?infotype=SA&subtype=WH&htmlfid=SWW14019USEN#loaded.

[18] O. H. Ibarra and C. E. Kim. Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *J. ACM*, 22 (4), 1975.

[19] K. Iwama and S. Taketomi. Removable Online Knapsack Problems. 2380:293–305, 2002.

[20] S. Jiang and X. Zhang. LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *ACM SIGMETRICS*, 2002.

[21] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*, 1994.

[22] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

[23] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12), 2001.

[24] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SOCC*, 2014.

[25] S. Lightstone, M. Surendra, Y. Diao, S. S. Parekh, J. L. Hellerstein, K. Rose, A. J. Storm, and C. Garcia-Arellano. Control Theory: a Foundational Technique for Self Managing Databases. In *ICDE Workshops*, 2007.

[26] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, 2003.

[27] V. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing Buffer Pool Memory in Multi-tenant Relational Database-as-a-service. *PVLDB*, 8(7), 2015.

[28] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *ACM SIGMOD*, 1993.

[29] L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. *IEEE/ACM Trans. Netw.*, 8(2), 2000.

[30] R. P. Wooster and M. Abrams. Proxy Caching That Estimates Page Load Delays. *Computer Networks*, 29(8-13), 1997.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. NSDI, 2012.

[32] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX*, 2001.