Eagle-Eyed Elephant: Split-Oriented Indexing in Hadoop

Mohamed Y. Eltabakh[1*]
Fatma Özcan[2]
Yannis Sismanis[2]
Peter J. Haas[2]
Hamid Pirahesh[2]
Jan Vondrak[2]

[1]Worcester Polytechnic Institute
Worcester, MA, USA
meltabakh@cs.wpi.edu

[2]IBM Almaden Research Center
San Jose, CA, USA
{fozcan, syannis, phaas, pirahesh, jvondrak}@us.ibm.com

ABSTRACT

An increasingly important analytics scenario for Hadoop involves multiple (often ad hoc) grouping and aggregation queries with selection predicates over a slowly changing dataset. These queries are typically expressed via high-level query languages such as Jaql, Pig, and Hive, and are used either directly for business-intelligence applications or to prepare the data for statistical model building and machine learning. In such scenarios it has been increasingly recognized that, as in classical databases, techniques for avoiding access to irrelevant data can dramatically improve query performance. Prior work on Hadoop, however, has simply ported classical techniques to the MapReduce setting, focusing on record-level indexing and key-based partition elimination. Unfortunately, record-level indexing only slightly improves overall query performance, because it does not minimize the number of mapper "waves", which is determined by the number of processed splits. Moreover, key-based partitioning requires data reorganization, which is usually impractical in Hadoop settings. We therefore need to re-envision how data access mechanisms are defined and implemented. To this end, we introduce the Eagle-Eyed Elephant (E3) framework for boosting the efficiency of query processing in Hadoop by avoiding accesses of data splits that are irrelevant to the query at hand. Using novel techniques involving inverted indexes over splits, domain segmentation, materialized views, and adaptive caching, E3 avoids accessing irrelevant splits even in the face of evolving workloads and data. Our experiments show that E3 can achieve up to 20x cost savings with small to moderate storage overheads.

1. INTRODUCTION

This work was motivated by a desire to improve the performance of massive-scale analytics in an important and increasingly common user scenario. Specifically, we focus on warehouse-like exploratory analysis environments in which the Hadoop [13] open source implementation of MapReduce [7] is used together with a higher-level interface such as Jaql [3], Pig [22], or Hive [30] to execute multiple (and often ad hoc) grouping and aggregation queries with selection predicates over a slowly changing dataset. The queries are used either directly for business-intelligence applications or to prepare the data for statistical model building and machine learning. The query workload typically evolves in ways that are not known a priori; i.e., certain attributes and attribute values are intensely queried for a while, after which a different set of attributes and attribute values are frequently queried, and so on, but the precise sequence of query patterns is not known in advance.

[*]The author conducted most of this work at the IBM Almaden Research Center.

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy.

1.1 Motivating examples

One real-world example of such a scenario is provided by the authors' experience with an established credit card company having leading analytics capabilities for predicting credit card fraud, as well as strong data-warehouse capabilities for analyzing information generated from cardholder accounts, merchants, consumer banks, and more. Hadoop is used to extend the customer's analytics capabilities because it provides a scalable and cost-effective platform for storing many years of highly-structured credit-card transaction data. The analytics needed for marketing and risk assessment are highly varied, involving custom algorithms and many different third-party modules that require exploratory analysis and various statistical summaries. Multiple selective queries need to be run on the transaction data in order to support the development of new prediction techniques, e.g., as new fraud patterns emerge. Specifically, queries are used to repeatedly sample various portions of the data for building, refining, and validating models. Queries are also used to materialize data representing different time periods, customer segments, and so on, to feed into the models once they are built.

Another example is provided by our collaboration with a leading wind-power generation company. Simulating global wind patterns is critical for the company's business and produces petabytes of semi-structured and nested data that are stored in a Hadoop cluster. The company's scientists need to explore the simulation output by repeatedly running ad hoc selection queries, continuously evaluating and analyzing the results.

As illustrated by these examples, the data may be structured or only semi-structured. Hadoop is well suited to processing many types of datasets, avoiding the burdensome need to structure the data into rigid schemas prior to analysis. This flexibility, along with attractive price-performance characteristics and low startup costs, helps explain Hadoop's growing popularity.

1.2 The Need for Indexing in Hadoop

Despite Hadoop's excellent scalability properties, users have recognized that, as with traditional database systems, there is continuing pressure to enhance query processing efficiency. Indeed, evolving infrastructure-as-a-service (IaaS) pricing models require users to pay according to the hardware and energy resources that they use [14], so there is an increasing need to reduce such costs.

The most important strategy for improving query performance is to avoid processing data that is irrelevant to the query. Prior work on selective processing in Hadoop, however, has simply ported classical techniques for this task directly to the MapReduce setting, focusing on record-level indexing and key-based partition elimination. Neither of these naive approaches are very effective.

The key issue for indexing is that the wall-clock execution time for a query depends on the number of waves of mappers that are executed, which in turn depends on the number of processed *splits*—logical partitions of the data as specified in the Hadoop InputFormat interface. (One mapper task is created for each split.) The cost of processing a split is dominated by the I/O cost of reading the split and the overhead of starting up the mapper that processes the split. So the cost of processing one or two records in a split is typically not much less than the cost of processing every record in the split. Traditional record-level indexing techniques that ignore this feature of Hadoop do not perform well. Indeed, previous work [9] has demonstrated that such techniques do not translate to end-to-end savings unless both the Hadoop Distributed File System (HDFS) and Hadoop itself are thoroughly re-engineered; our goal is to improve the performance of Hadoop as-is.

1.3 The Need for Improved Indexing

In more detail, prior indexing schemes [1, 8, 16] consider only clustered indexes on the keys used to partition the data into splits. Such indexes do not suffice in themselves. For example, suppose that the data are partitioned into splits by time and that transactions in each split are sorted by their timestamps. Consider the query that returns all transactions between Jan 2008 and Aug 2008 for a given customer at a given store. Using a clustered index on dates, we can *eliminate* (i.e., avoid the reading and processing of) splits that do not belong to this time interval. If, however, the customer only shops at the store once per month, Hadoop may still process many splits that contain no transactions for that customer. In this case, a secondary index on customer IDs would eliminate many splits. Similarly, if the store opened in Jul 2008, then a composite index on (store, date) value pairs would be more effective than just the clustered index on dates. In our setting, where the workload is generally unknown a priori, there is no guidance on which indexes to construct. For example, the TPoX and TPCH benchmark datasets considered in Section 4 contain 46 and 7 attributes, respectively, that each have the potential to eliminate 80% of the splits in a query. A partitioning-key approach would essentially ignore all of these attributes save one, thereby foregoing a wide range of split-elimination opportunities.

In a similar manner, attempts at directly porting classical partition-elimination methods to the Hadoop setting also have limited effectiveness. Classical approaches are based on physically partitioning the data according to some key, and then avoiding processing those partitions not referenced by the query. The Hive interface [30] provides this functionality, but requires an expensive map-reduce job to partition the data (unless the data is already partitioned by the key), because HDFS does not provide any direct user mechanisms for data placement. If a good partitioning key is not known in advance—the usual case in our scenario—then reorganizations may need to occur frequently, adding enormous overhead to query processing.

Further complicating attempts at both indexing and partition elimination is the fact that the nature of data evolution in Hadoop differs from that in a classical database system. Specifically, HDFS stores large files as a series of blocks distributed over a cluster of data nodes and replicated for purposes of fault tolerance [29]. HDFS does not directly support in-place updates or appends to existing files; instead, new data (in the form of files) rolls in and old data is rolled out by deleting or archiving files.

1.4 E3 Overview

Clearly, we need to re-envision how data access mechanisms are defined and implemented in Hadoop. To this end, we introduce the Eagle-Eyed Elephant (E3) framework for boosting the efficiency of query processing in a Hadoop/Jaql processing environment.[1] The key idea behind E3 is to focus directly on elimination of splits, using enhanced, split-oriented indexing techniques that are coarser, but more effective, than record-based indexing. Also, in contrast to prior schemes that only index on partitioning keys, E3 provides a more flexible indexing scheme, based on our observation that non-key fields having large numbers of distinct values can be very effective for split elimination.

Specifically, E3 operates over a "flattened JSON" view of the data in a file (see Section 2) in which each file and split consists of arrays of *records*. Each record is a collection of *fields*, where a field consists of a (name, atom) pair. For purposes of this paper, an *atom* is an elementary data value that can be a string, date, boolean, or number. (As discussed in Section 2, the file of interest need not actually be stored in JSON format.)

E3 computes certain split-level statistics for each field—such as min, max, and ranges for a numerical field—and stores them in a "range index". These range statistics, computed via a "domain segmentation" scheme, allow the system to effectively prune splits that do not satisfy given range predicates in a query of interest.

E3 also automatically builds a split-level inverted index over all string fields. Importantly, all fields are indexed, and not just the partitioning field(s); as indicated above, such comprehensive indexing can greatly increase opportunities for split elimination. Moreover, there is no need to guess a partitioning key or to repeatedly repartition the data as the workload changes; E3 simply creates an index on all fields once, and amortizes the creation cost over multiple queries. The only system resource needed in addition to Hadoop is a lightweight cataloging service to maintain the metadata.

One challenge to the inverted-index scheme is the presence of "nasty" atoms, that is, values that are globally infrequent but appear once or twice in a large number of splits. E3 can automatically identify these values and enhance the inverted index with materialized views over the data records having nasty atoms. Then, given a query predicate that references a nasty atom, only a small number of splits in the materialized view need be accessed, rather than a large number of splits in the original file.

A similar problem is posed by conjunctive queries that reference "nasty" atom pairs, where each atom in the pair appears in many splits but the atoms appear jointly in only a few records. Consider, e.g., (ship date, delivery date) atom pairs. A given individual ship date may appear in many transactions, and similarly for a given delivery date, but if these dates are unusually close or far part, the number of actual transactions that contain both dates simultaneously will be small. Substitute products are another example: many people buy iPhones or Droids, but few people buy both together. Because of the prohibitive quadratic number of atom pairs, a complete index for all atom pairs is too expensive to compute and store. Hence E3 employs a novel main-memory adaptive caching method to maintain an inverted index only of those nasty atom pairs that are the most "valuable" in that they are referred to frequently and result in the elimination of many splits when cached.

E3 automatically combines the foregoing indexes and materialized views at query execution time to access the minimal set of splits required to process the query. E3 avoids the need for user-specified physical design, data movement or reorganization, or a priori knowl-

[1] Although we focus on queries expressed in the Jaql language, our techniques can be applied to queries in Pig, Hive, and so on.

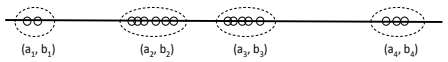


Figure 1: Creating clustered ranges.

edge about the query workload (though workload information can be exploited if available). Experiments using data with hierarchical and flexible schemas, as well as data with flat and fixed schemas, demonstrate savings in query response time of up to 20x due to split elimination while requiring a small to moderate storage overhead (5%–21%) for the corresponding indexes and materialized views. Our indexing techniques are potentially interesting even in the setting of traditional parallel databases [23]: these systems typically use only partitioning attributes for "data localization" (i.e., partition elimination).

The rest of the paper is organized as follows. In Section 2, we describe the range index, inverted index, materialized views, and caching algorithm. In Section 3, we discuss how these indexing techniques are used collectively to eliminate splits at query time. Section 4 describes our experimental study, Section 5 reviews related work, and Section 6 concludes the paper.

2. EAGLE-EYED ELEPHANT (E3)

In this section, we describe the core techniques used in E3 to speed up queries with selection predicates by avoiding useless splits to the greatest extent possible. Sections 2.1 and 2.2 discuss the computation of the range index and the inverted index. Section 2.3 describes the identification of nasty atoms and the use of a materialized view to tame them. Section 2.4 describes the main-memory adaptive caching algorithm used to handle nasty atom pairs.

As indicated previously, E3 can operate over a wide range of storage formats. Indeed, because E3 uses Jaql for processing, it can exploit the adapters and converters provided by Jaql for translating the actual storage format into a JSON view for processing. Formats that E3 can handle include CSV files and HDFS sequence files. In more detail, E3 operates on JSON views generated by Jaql *file descriptors* [3]. A file descriptor specifies the specific adapters and converters used for the data, as well as the InputFormat used for creating splits. Note that E3 may generate different indexes on the same data if different file descriptors are used. Therefore, for each file in the warehouse, E3 maintains both a file descriptor and file signature in the system catalog to ensure that the correct index is used to eliminate splits in a Jaql query over the file. E3 actually uses a "flattened" JSON view of the data. In general, JSON data comprises atoms, arrays, and records (sets of name-value pairs), which can be arbitrarily nested to any depth. E3 flattens this nested structure so that each field name becomes a root-to-leaf path name and each field value corresponds to the atom at the end of the path.

2.1 Domain Segmentation for Range Indexing

By creating a "range index" which contains, for each split, range statistics on each numerical and date field[2], we can quickly eliminate splits that do not satisfy a given range or equality predicate appearing in the query of interest. E.g., if we know that the minimum and maximum values of "weight" in a given split are 2 lbs and 3 lbs, respectively, then we know immediately that no records in the split satisfy a predicate of the form "weight between 1.2 lbs and 1.5 lbs". However, if the atoms within a split are clustered in small subranges of the overall min-max range, then merely storing the minimum and maximum values for a field may lead to many false positives. For example, suppose the atoms for a given field in a given split are as in Figure 1, so that the minimum and maximum atom values are a_1 and b_4. Then we would incorrectly assume that there may be records in the split that satisfy the predicate "value between x_1 and x_2" if $x_1 > b_1$ and $x_2 < a_2$.

E3 therefore uses a simple and fast one-dimensional "domain segmentation" technique that generates multiple ranges for each field in each split. Given a bound k on the number of ranges for each field, and a set of values $\{v_1, \ldots, v_n\}$, our goal is to compute at most k segments of the min-max interval, i.e., at most k ranges, such that each v_i is contained in some range and the ranges are configured "as tightly as possible" (see Figure 1).

The algorithm works as follows for a given split: We determine the distinct values among $v_1, \ldots, v_n$ and sort them. If the number of distinct values is at most k, we create a separate range $[v_i, v_i]$ for each distinct value (subject to the consecutive values optimization described below). Otherwise, we have $\ell > k$ distinct values, denoted in ascending order as $v_{\text{min}} = a_1 < a_2 < \cdots < a_\ell = v_{\text{max}}$. We compute the $\ell - 1$ consecutive gaps $g_i = a_{i+1} - a_i$, and sort these in descending order. Let $i(1), i(2), \ldots, i(k-1)$ denote the indices of the $k-1$ largest gaps. For each gap $g_{i(j)}$, we remove the interval $(a_{i(j)}, a_{i(j)+1})$ from $[v_{\text{min}}, v_{\text{max}}]$. What remains is a collection of k ranges—i.e., subintervals of $[v_{\text{min}}, v_{\text{max}}]$—of the form $[c_1, d_1], [c_2, d_2], \ldots, [c_k, d_k]$, where each c_k and each d_k is equal to one of the v_i's. These are the ranges returned by the algorithm. If a predicate does not hit any of the ranges, the split can be eliminated.

For integer-valued and date-valued fields, we actually use $a_{i+1} - a_i - 1$ rather than $a_{i+1} - a_i$ to define the size of the gap g_i. The former expression is the number of possible predicate values in the interior of the interval $[a_i, a_{i+1}]$. In particular, gaps between two consecutive integers are never selected, because there is no benefit in doing so. If we find that fewer than $k-1$ gaps have size $g_i = a_{i+1} - a_i - 1 > 0$, we stop and form exactly those ranges determined by the positive gaps.

Under a query workload having uniformly distributed equality and range predicates that hit the interval $[v_{\text{min}}, v_{\text{max}}]$, the foregoing scheme produces ranges that maximize the probability that a predicate will eliminate the split. To see this, assume without loss of generality that $v_{\text{max}} - v_{\text{min}} = 1$. First consider a random equality predicate $q_e(attr, w)$ that searches for records with attribute *attr* equal to w. The probability that the split is eliminated equals the probability P_{eq} that w lies in one of the $k-1$ selected gaps, where $P_{\text{eq}} = \sum_{j=1}^{k-1} g_{i(j)}$. Similarly, for a predicate $q_r(attr, w_1, w_2)$ that searches for all records having a value of *attr* between w_1 and w_2, the split is eliminated if both w_1 and w_2 lie in the *same* gap, which happens with probability $P_{\text{range}} = \sum_{j=1}^{k-1} g_{i(j)}^2$. Because we selected $g_{i(1)}, \ldots, g_{i(k-1)}$ to be the $k-1$ largest gaps, it follows that, under the condition that we have at most k ranges, our choice of ranges simultaneously maximizes P_{eq} and P_{range}.

If there happens to be workload information available, we can exploit it simply by redefining each g_i as the fraction of historical predicate values that have fallen in the interval (a_i, a_{i+1}); thus g_i estimates the probability that a future value will fall in this interval. Both the segmentation algorithm and optimality argument remain essentially unchanged.

2.2 Inverted Index

E3 builds an inverted index over each string field[3] in the data file and uses the index when evaluating equality predicates. The index

[2]We can compute range statistics on any data type that has a well-defined sort order, including strings. In the current paper, we focus on numeric and date fields.

[3]It is also possible, for example, to build the inverted index over strings, dates, and numbers, and use the inverted index in lieu of the range index; we focus on the string-only case in the current paper.

is implemented using fixed-length bitmaps, where the number bits in each bitmap equals to the number of splits in the dataset. T bitmap for a given atom v has its ith bit set to 1 if and only if t ith data split contains v. To compactly represent the index, ea bitmap is compressed using Run-Length Encoding (RLE) [12]. our experiments show, RLE compression is very effective, reduci the index size by up to 85%.

E3 constructs the inverted index by performing a single ma reduce job over the data. The map phase reports each atom v a the split ID in which v appears. The reduce phase then groups the output records according to v and merges the split IDs to form t bitmap array, which is then compressed using RLE.

2.3 Materialized View

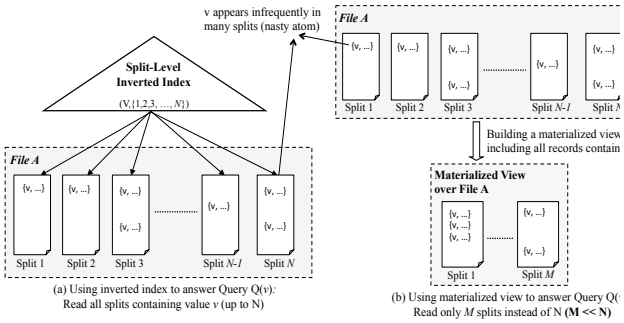


Figure 2: Inverted index vs. materialized view.

Figure 2 gives an example of a nasty atom v that appears once or twice in each of the N splits of "File A". If we use an inverted index—see Figure 2(a)—all of these splits will be accessed whenever an equality predicate involving v is evaluated. However, by building a materialized view on File A that stores all records containing v, queries involving v can be answered by processing the M splits of the materialized view instead of the N splits of the original data, where $M << N$; see Figure 2(b).

The number of nasty atoms in a dataset can be very large and the allotted space for the materialized view is expected to be very small, e.g., 1% or 2% of the data size. Thus the key challenge is to choose the "best" subset of nasty atoms to store in the materialized view. We formulate this optimization problem as a submodular knapsack problem and provide a practical approximate solution.

Denote by M and R the maximum number of splits and records that can fit in the materialized view. Also let splits(v) and records(v) be the set of splits and set of records containing a given atom v. Denote by $|\text{splits}(v)|$ and $|\text{records}(v)|$ the cardinalities of these sets. Each atom v that we consider for the materialized view has a "profit" defined as $\text{profit}(v) = |\text{splits}(v)| - M$, representing the number of splits saved due to reading splits in the materialized view instead of the original data splits containing v. For a set V of atoms considered jointly for inclusion in the materialized view, we define the overall profit as $\text{profit}(V) = \sum_{v \in V} p(v) \cdot \text{profit}(v)$, where $p(v)$ is the probability that a query will contain an equality predicate that references atom v. The probability distribution p can be estimated from the query workload if available, and thus giving greater weight to frequently referenced atoms. If the query workload is unknown, then we assume a uniform query distribution over all atoms in the dataset; in this case $p(v)$ can be treated as a constant and effectively ignored in the analysis. In a similar manner, we can define cost(v) as the number of records that will be copied to the materialized view and thus use up part of its allotted space. (We use the number of records as a proxy for space consumed because it is extremely expensive to track the actual space consumption associated with each record.) Noting that the sets records(u) and records(v) can overlap for distinct u and v, we define the cost of storing a subset V of atoms in the materialized view as $\text{cost}(V) = |\bigcup_{v \in V} \text{records}(v)|$.

```
selectNastyAtoms(S, R, L, α)
Input:
    - Set S of n values, S = {v1, ..., vn},          // S contains all values in the dataset
    - Maximum number of records in the materialized view R
    - Minimum number of splits a value may appear in L.
    - Overlapping factor α ≥ 1.
Output:
    - OutputList = {}                                // Output list containing the selected nasty values

(1) Compute U = R/L.                                 // U is the upper bound for the number of needed values
(2) Build TopU list ← Scan the values in S and keep only the top U values w.r.t. profit(v)/cost(v).
(3) Build SortedU list ← Sort the values in TopU list in descending order w.r.t. profit(v)/cost(v).
(4) Define totRecords = 0.
(5) For (i ←1 to U) Loop
(6)     - Let v = SortedU(i).
(7)     - Add v to OutputList.
(8)     - totRecords += |records(v)|.                //Add the number of records containing v to the counter
(9)     - If (totRecords ≥ R * α ) Then              // If the number of records exceeded the materialized
(10)        - Exit the loop.                         //    view capacity, then exit the loop.
(11) Return OutputList
```

Figure 3: Modified greedy algorithm for selecting the materialized view atoms.

The optimization problem is therefore to choose a subset of atoms V to maximize profit(V) subject to the constraint that cost$(V) \leq R$. The function cost(V) is "submodular" and hence our optimization problem is a special case of the "submodular knapsack" problem. Unfortunately, this problem is very hard even to approximate: it is known that there is no approximation better than a multiplicative $\sqrt{n/\log n}$ factor [28]. Furthermore, our problem contains as a special case the "densest k-subgraph problem", for which no practical approximation algorithms are known. To develop a practical solution for constructing the materialized view, we ignore the overlap among record sets and approximate (actually, overestimate) the cost as $\text{cost}(V) \approx \sum_{v \in V} \text{cost}(v) = \sum_{v \in V} |\text{records}(v)|$. In this case, the submodular knapsack problem reduces to a classical 0-1 knapsack problem. Our experiments indicate that this approach yields good results in practice (see below), especially since the space constraints in our problem are not hard constraints—i.e., using slightly more than (or fewer than) M splits is acceptable.

Even the simplified 0-1 knapsack problem is NP-complete [11], but efficient approximation algorithms exist. It is known that a $(1 - \epsilon)$-approximation can be found in time polynomial in $1/\epsilon$ and the number of elements (an "FPTAS" [15]). For our implementation, since we are dealing with massive datasets, we opt for a simple greedy algorithm that provides a 0.5-approximation in the worst case [6]. (Actually, for settings like ours, where the costs are very small compared to the capacity of the knapsack, the greedy algorithm performs much better.) In the following, we present our modifications to this algorithm to efficiently construct a materialized view over a given dataset in a scalable manner; see Figure 3 for pseudocode.

The naive greedy algorithm sorts all atoms in the dataset with respect to decreasing profit(v)/cost(v) ratio. Then it selects atoms from the top of the sorted list and adds their corresponding records to the materialized view until it is full. This naive algorithm needs to be modified because (1) sorting all atoms in the dataset is very expensive especially when performed over the large-scale data manipulated by Hadoop, and (2) building the materialized view incrementally, e.g., by adding the records corresponding to a selected atom in each step, is infeasible in the Hadoop system since HDFS does not support random I/O over the data. To avoid sorting all the

atoms in the dataset, we compute an upper bound U on the number of atoms that can possibly fit in the materialized view (see below). This upper bound is typically multiple orders of magnitude smaller than the total number of atoms in the dataset. Given U, we perform a scan over the atoms and keep the top U atoms with respect to their profit(v)/cost(v) ratio (Step 2 in Figure 3). This step can be performed using a min-heap structure without any sorting. Finally, we sort the U atoms according to decreasing profit(v)/cost(v) ratio (Step 3), and then select atoms from the top of the list until the materialized view is full (Steps 5–10). A sorting step is still needed because U is an overestimate, but its cost is significantly lower than that of sorting all the atoms.

The upper bound U is computed as follows. We first introduce a lower bound L that represents the minimum number of splits in which an atom v must appear to be considered as a candidate for the materialized view. That is, if v appears in less than L splits, then its corresponding records are not deemed worthy of being accepted into the materialized view. Typically, L is defined as a multiple of the materialized view size, e.g., the default value for L in E3 is $3M$; where M is the materialized view size in splits. Since an accepted atom v will appear at least once in each of L splits, it follows that v will contribute at least L records to the materialized view. The materialized view can hold at most R records, and thus an upper bound on the number of atoms that can fit in the materialized view is $U = R/L$ (Step 1 in Figure 3).

Because the materialized view cannot be built incrementally, the modified greedy algorithm merely reports the list of chosen atoms that will contribute to the materialized view (Step 7 in Figure 3). As each atom v in the sorted list is added to the materialized view, a counter is incremented by $|\text{records}(v)|$; the counter represents the amount of allotted space used up so far. Notice that, in Step 9, the algorithm multiplies the maximum capacity of the materialized view (R) by an *overlapping factor* α, where $\alpha \geq 1$, to heuristically account for the possible overlapping among record sets. In our experimental analysis we observed that the overlapping among record sets does not significantly affect the space utilization of the materialized view; e.g., when α is set to 1 (thus ignoring overlaps), the materialized view space is still approximately 95% utilized.

To create the materialized view for a given dataset, we execute a map-reduce job followed by a map-only job. In the map-reduce job, the map phase reports each atom v along with the split ID in which v appears and the number of records containing v in that split. The reduce phase groups the output records according to v, computes the total number of splits and records in which v appears, i.e., $|\text{splits}(v)|$ and $|\text{records}(v)|$, and then executes the greedy algorithm of Figure 3. The map-only job then scans the data and copies the records containing the chosen atoms to the materialized view.

2.4 Adaptive Caching

We now consider the issue of nasty atom pairs in which the individual atoms are frequent and appear in most (or all) of the splits, but the atom pairs appear jointly in very few records, and hence in very few splits. This situation differs from the one handled by the materialized view in the previous section, which helps only with atoms that appear in many splits but infrequently overall. In the case that we discuss here, individual atoms might well appear frequently and hence not appear in the materialized view.

For example, Figure 4 shows two atoms v and w such that each atom is frequent and appears in all splits, but the atom pair (v, w) appears in only one split (Split 3). If we try to use the inverted index of Section 2.2 to find the splits containing (v, w) by computing splits(v) and splits(w) and then intersecting these two sets—

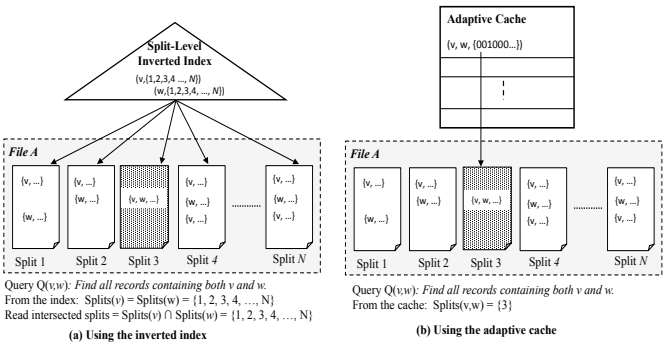


Figure 4: Inverted index vs. adaptive cache.

which is the only strategy available in the absence of any other information—then most of the returned splits will be false positives; see Figure 4(a). In the rest of this section, we focus on processing queries, denoted by $Q(v, w)$, that return all records containing both u and w; this query will appear as a subquery of any Jaql query with a conjunctive selection predicate that references both v and w. (E3 processes conjunctive queries involving more than two atoms by processing the atoms in a pairwise fashion and then intersecting the resulting sets of splits.) Motivated by our example, we define an atom pair (v, w) to be a *nasty* atom pair if $S_{(v,w)} > \theta$, where $S_{(v,w)} = |\text{splits}(v) \cap \text{splits}(w)| - |\text{splits}(v, w)|$. Here splits$(v, w)$ denotes the set of those splits that contain the pair (v, w) in at least one record and θ is a (large) user-defined threshold. The quantity $S_{(v,w)}$ is precisely the number of false positives obtained when split lists for individual atoms v and w are intersected in an attempt to find those splits that contain the pair (v, w). Equivalently, $S_{(v,w)}$ is the potential savings in split accesses if the pair (v, w) is cached.

Traditional database systems address this issue by building composite or multi-dimensional indexes, where the attribute combinations to be indexed are typically user-defined and are derived from workload and schema information. These approaches are not feasible for our Hadoop scenario, because the query workload and (possibly loose) schema are typically unknown a priori—indeed, users may not even know the fields in the dataset beforehand. Moreover, algorithms for exploring the space of all possible atom pairs to index are prohibitively expensive in large-scale data sets because they are inherently quadratic in time and space. Moreover, sampling and sketching techniques as in [5] are ineffective in finding nasty pairs because pairs are infrequent.

We propose an adaptive main-memory caching technique in which the system monitors the query workload and the atom pairs that are being queried, and then caches the pairs that are (1) recently queried, (2) frequently queried, and (3) nasty in the sense that $S_{(v,w)} > \theta$. As illustrated in Figure 4(b), the cache maintains atom pairs along with a bitmap for each atom pair specifying the splits that contain the pair. As shown in the figure, the cache is significantly more efficient than the inverted index in answering query $Q(v, w)$.

We must monitor the query workload and compute the actual splits containing each queried atom pair, i.e., compute splits(v, w) for each queried pair (v, w). To this end, we use a standard Hadoop counter (*Task.Counter.MAP_OUTPUT_RECORDS*) to identify the map tasks that have produced at least one tuple; the corresponding splits comprise splits(v, w). We can then perform the steps in Figure 5 to execute a given conjunctive query $Q(v, w)$ and to decide whether or not (v, w) is candidate for caching.

By monitoring the map tasks that produce output, the system determines the ID of every split that contains the (v, w) pair in at least

(1) splits(v) ← Probe the inverted index by value *v* to get its set of split Ids
(2) splits(w) ← Probe the inverted index by value *w* to get its set of split Ids
(3) splitsToQuery(v,w) ← splits(v) ∩ splits(w) // The set of intersected splits
(4) Start the query (job) execution over the set of splits in splitsToQuery(v,w)
(5) splits(v,w) ← Identify the split Ids assigned to map tasks that produced output
(6) $S_{(v,w)}$ = |splitsToQuery(v,w)| - |splits(v,w)| // Number of false-positive splits
(7) **If** ($S_{(v,w)} > \theta$) **Then**
(8)

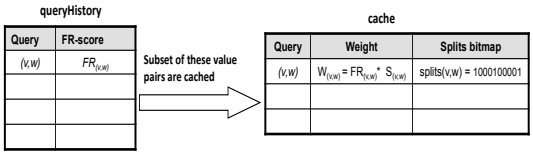


Figure 6: SFR data structures.

one record (Step 5 in Figure 5). If the number of the false-posi splits $S_{(v,w)}$ is greater than the user-defined threshold, then $(v$ is considered for caching by calling the insertCache() algori (Steps 6–8 in Figure 5). Note that these steps are performed c if an initial probe of the cache does not return a split list for pair (v, w), because the pair is not in the cache. The algori probeCache(v, w) used for this initial probe is described below.

As queries are issued over time, more and more candidate pairs are considered for caching. Because the cache size is limited, an efficient replacement policy is needed to keep the most promising pairs in the cache. Although an LRU replacement policy is easy to implement and maintain, our experiments show that LRU does not perform well for many query workloads. The main reason for this poor performance is that LRU takes into account only how recently an atom pair has been queried, ignoring other important factors such as the potential savings in splits if the pair were to be cached. In the following, we present the *SFR* (Savings-Frequency-Recency) cache replacement policy that we use in E3.

The SFR policy maintains in the cache the atom pairs having the largest "weights," where the weight of a pair (v, w) depends on (1) the potential savings in splits due to caching the pair, i.e., $S_{(v,w)}$, (2) the historical frequency with which (v, w) has been queried, and (3) how recently (v, w) has been queried. By taking the savings factor into account, SFR is able to catch the "precious" atom pairs that yield high savings and cache them for a while even if they are not queried extremely frequently. Reserving space for such infrequent but valuable atom pairs can significantly increase the overall effectiveness of the cache.

SFR maintains two data structures as shown in Figure 6, *query-History* and *cache*. The queryHistory structure maintains a log of queried atom pairs. Stored along with each pair is a partial weight, called the *FR-score*, that reflects both the frequency and recency of queries involving the pair. The FR-score does not take the potential savings into account. A small subset of these pairs are also stored in the cache structure; any pair stored in the cache structure also appears in queryHistory. E3 does not allow the queryHistory table to grow indefinitely. The system allows entries with FR-scores below a user-defined threshold to be periodically pruned, and a timestamp can be used to eliminate entries that have not been updated for a long period of time.

The cache structure maintains a list of nasty pairs. The final weight—which combines the FR-score and the potential savings in split accesses—is stored for each pair, along with a bitmap identifying the splits that contain the pair. The final weight for a pair (v, w) is obtained by multiplying the FR-score for (v, w) by the savings

insertCache ((v,w), splits(v,w), $S_{(v,w)}$)
Input:
- (v,w): The value pair to cache
- splits(v,w): Set of splits containing (v,w)
- $S_{(v,w)}$: The number of splits when caching (v,w)

Output: None

Global Parametrs:
- Recency factor ρ = 1
- Amplification factor λ (λ > 1)

(1) Increase ρ = ρ x λ // Amplify the recency factor

// Update the queryHistory information
(2) **If** (v,w) is in queryHistory **Then**
(3) - $FR_{(v,w)}$ += ρ // The addition op. gives higher weight for frequent pairs
(4) **Else**
(5) - Insert ((v,w), $FR_{(v,w)}$ = ρ) into queryHistory

// Insert into the cache
(6) $W_{(v,w)} = FR_{(v,w)} * S_{(v,w)}$ //Compute the final weight of (v,w)
(7) **If** cache is not full **Then**
(8) - Insert ((v,w), $W_{(v,w)}$, splits(v,w))
(9) **Else**
(10) - $W_{(v',w')}$ ← Find the entry with the smallest weight, say for (v',w')
(11) - **If** ($W_{(v',w')} < W_{(v,w)}$) **Then**
(12) - Replace the entry of (v',w') with the entry of ((v,w), $W_{(v,w)}$, splits(v,w))

Figure 7: Inserting into the cache: insertCache() algorithm.

probeCache ((v,w))
Input:
- (v,w): The probing value pair

Output:
- List of splits to query (if cache hit) or Null (if cache miss)

Global Parametrs:
- Recency factor ρ = 1
- Amplification factor λ (λ > 1)

(1) **If** (v,w) is cache hit **Then**
(2) - Increase ρ = ρ x λ // Increase the recency factor
(3) - $FR_{(v,w)}$ += ρ // Update FR-score in queryHistory
(4) - $W_{(v,w)} = FR_{(v,w)} * S_{(v,w)}$ // Update the weight in the cache
(5) - **Return** splits(v,w) from the cache
(6) **Else**
(7) - **Return** Null

Figure 8: Probing the cache: probeCache() algorithm.

factor $S_{(v,w)}$.

Figure 7 displays the cache insertion algorithm, insertCache(), for inserting a new (v, w) pair. Conceptually, with the arrival of each new pair, the FR-score of all items maintained in queryHistory decays in order to indicate that these items are getting older and to put more emphasis on the new pairs. Such decay could be implemented by multiplying the FR-score of every pair in queryHistory by a decay factor (having a value less than 1) every time a new atom pair arrives. Because it is very expensive to perform this operation every time a pair arrives, we implement the same idea equivalently by applying a *recency factor* ρ to the new pair while keeping the other items in queryHistory unchanged. With every new pair, the recency factor gets amplified by a factor of $\lambda > 1$ (Step 1 in Figure 7). Then the algorithm updates the information in queryHistory by increasing the FR-score $FR_{(v,w)}$ by ρ if the pair already exists, or inserting a new record if the pair does not exist (Steps 2–5 in Figure 7). Notice that the addition operation in Step 3 ultimately gives large weights to frequently queried pairs. To insert (v, w) into the cache, the final weight $W_{(v,w)}$ is first computed (Step 6 in Figure 7). Next, a new entry containing information for (v, w) is inserted into the cache if the cache has available space (Step 8 in Figure 7); otherwise the algorithm finds the entry with the smallest weight, say $W_{(v',w')}$, and replaces this entry with the new one if $W_{(v,w)} > W_{(v',w')}$ (Steps 10–12 in Figure 7). Because the recency factor ρ gets amplified with every new pair, it will eventually overflow. To avoid this problem, the system periodically—e.g., after every k new arrivals—normalizes all the scores while preserving their relative orders.

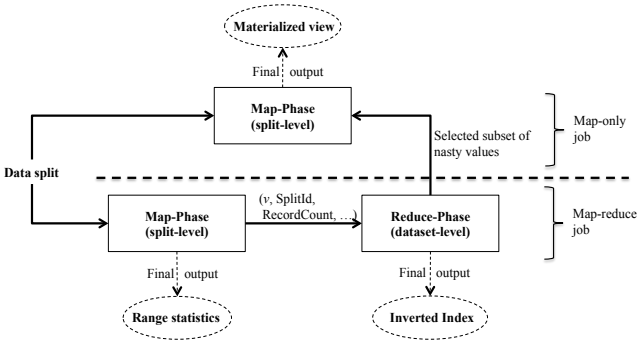


Figure 9: Computational flow for building E3 data structures.

evaluatePredicate (file, P)
Input: P = a predicate with conjunctions and disjunctions
file = the file to query
Output: List of split IDs to process

(1) **If** P contains at least one disjunction **Then**
(2) convert P into disjunctive normal form: $D_1 \vee D_2 \vee \ldots \vee D_n$
(3) **Else** $D_1 = P$
(4) **For each** D_i **Loop**
(5) {List(D_i), useMV(D_i)} = evaluateConjunct'(file, D_i)
(6) **If** useMV(D_i) = true for all $1 \leq i \leq n$ **Then**
(7) **Return** MV
(8) **Else Return** $\bigcup_{1 \leq i \leq n} List(D_i)$

Figure 10: Split list computation with disjunctions.

The probeCache() algorithm mentioned previously is given in Figure 8 and is a simplified version of the insertCache() algorithm. If the cache is probed with a given pair (v, w) and the pair is in the cache (a cache hit), then the algorithm updates the $FR_{(v,w)}$ and $W_{(v,w)}$ scores in queryHistory and cache, respectively, and returns a list of split IDs corresponding to that pair, i.e., splits(v, w) (Steps 2-5 in Figure 8). Notice that if (v, w) is in the cache, then it is guaranteed to have a corresponding record in the queryHistory table. If the probe results in a cache miss, then the algorithm returns *Null* (Step 7 in Figure 8).

SFR is sensitive to the amplification factor λ. The larger the value of λ, the more emphasis is added to the recency factor against the frequency and savings factors (and vice versa). For example, as λ gets larger, the performance of SFR gets closer to that of LRU. In our experimental analysis (Section 4), we study various values of λ and their effect on the cache performance.

2.5 Computational Flow

To efficiently build the ranges, indexes, and materialized view without unnecessary scans over the data, E3 shares the map and reduce tasks whenever possible among the different computations. In Figure 9, we present a flow diagram of the E3 computations. These computations require only two passes over the data. In the first pass (lower half of the diagram), a map-reduce job is performed in which the map phase computes the range statistics (Section 2.1) and reports each atom v along with its split ID and the number of records in the split that contain v. The reduce phase groups the records generated during the map phase according to v, builds the inverted index (Section 2.2), and applies the modified greedy algorithm in Figure 3 to identify the subset of nasty atoms to be included in the materialized view. In the second pass (upper half of the diagram), a map-only job is executed that stores in the materialized view every data record that contains one or more of the selected nasty atoms (Section 2.3). Importantly, any of the indexing components can be turned on or off using boolean flags, depending on user requirements. For example, if the data will not be queried frequently enough, then users may not be interested in paying the cost of building the materialized view. In this case, the analysis tool automatically bypasses the execution of the modified greedy algorithm in the reduce phase and also skips the second map-only job.

3. USING THE INDEXES AT QUERY TIME

We now describe how the range and inverted indexes, materialized view, and adaptive cache are used to compute a minimal list of splits to be processed for a query with selection predicates. E3 stores the materialized view in HDFS using the same format as the original data file. There are several options for storing the indexes and the statistical information that is computed for each split. They can be stored in a central repository or distributed and stored together with their corresponding splits. As mentioned previously, Hadoop has significant overheads for creating map tasks. To avoid such overheads, it is important to eliminate the splits before the map tasks are created. Therefore, E3 stores the range and inverted index in a central repository that is queried once by the Hadoop InputFormat process before Hadoop's JobTracker creates the required mappers and assigns splits to each mapper. In E3, we chose an RDBMS as the repository. In general, we can use any (possibly lightweight) repository that supports indexes.

E3 takes as input a conjunctive predicate of the form $P = p_1 \wedge p_2 \wedge \ldots \wedge p_n$ and a file name, and returns a list containing the IDs of the splits in the file that must be processed in order to evaluate the predicate P. E3's algorithm for this task is invoked from the InputFormat, and as a result can be used with any high-level query language on Hadoop, including Hive, and Pig, as well as incorporated into plain map reduce jobs. (With a slight abuse of notation, we denote by MV both the materialized view and the list of splits in the materialized view.) The algorithm first initializes *List*, the list of split IDs, to contain the IDs of every split in the file, thus representing the set of splits that must be processed in the absence of additional information. For equality predicates, the algorithm uses the range index for date and number fields, the inverted index for string fields, and the adaptive cache; for non-equality predicates, the algorithm uses the range index alone. (Here $=$ and *in* are considered to be the equality predicates.) The final list of splits to process is the intersection of the split lists for all of the p_i's, since predicate P is a conjunction. If any one of the equality predicates in P can be applied using the materialized view MV, then we compare the size of the final split list with the number of splits in MV, and choose the option with the smaller number of splits.

In the general case where the query contains both conjunctions and disjunctions, we convert the query predicate into disjunctive normal form, execute the above algorithm on each of the conjuncts, and finally union the resulting split lists to compute the final list of split IDs to process. Some care must be taken, however, in the presence of the materialized view. Specifically, if the algorithm returns split IDs from MV for at least one conjunct D_i, then it is possible to access a given record twice, once from MV when processing D_i and once from the original data file when processing another conjunct D_j for which MV is not used. To avoid this problem, we modify the previous algorithm to return for each conjunct D_i the split list that is computed without MV usage, denoted by List(D_i), and a boolean, denoted by useMV(D_i), that indicates whether or not MV could possibly be used to process D_i. The final algorithm is provided in Figure 10. It returns the split IDs in MV if and only if MV can be

used for *all* of the disjuncts (Steps 6-7); otherwise, the algorithm returns the union of the split lists for all disjuncts (Step 8).

4. EXPERIMENTS

We empirically evaluated E3's effectiveness in speeding up analytical queries having selection predicates. We studied (1) the wall-clock savings in query response times, (2) the computation costs of building the range statistics, inverted index, and materialized view, (3) the storage overhead of the indexes and materialized view, and (4) the effectiveness of the caching techniques. Overall, the experiments show that E3 yields significant speedups in query response times compared to plain Hadoop, while avoiding unreasonable overheads.

In the experiments, we built the range index over all numeric and date fields and built the inverted index over string fields. The target maximum size M of the materialized view was set to 1% of the original data size. The lower bound L in Figure 3 was fixed at its default value ($L = 3M$). Changing the value of L is expected to have a minor impact on overall performance: it affects only the number of candidate nasty atoms to be sorted in memory before selecting the final set of atoms for insertion into the materialized view, and the in-memory sorting cost is typically negligible with respect to the overall construction cost

Cluster Setup. The experiments were evaluated on a 41-node IBM SystemX iDataPlex dx340. Each server comprised two quad-core Intel Xeon E5540 64-bit 2.8GHz processors, 32GB RAM, and 5 SATA disks, with a 1GB Ethernet interconnect. Each server ran Ubuntu Linux (kernel version 2.6.32-24), IBM Java 1.6, Hadoop 0.20.2. Hadoop's master processes (MapReduce JobTracker and HDFS NameNode) were installed on one server and the remaining 40 servers were used as workers. Each worker was configured to run up to four map and four reduce tasks concurrently. The following configuration parameters were overridden in order to boost performance: sort buffer size was set to 512MB, replication factor was set to 2, JVM's were re-used, speculative execution was turned off, and a maximum of 5GB JVM heap space was used per task. We repeated each experiment 3 times and report the average of the results.

Datasets. We generated datasets from two different benchmarks: TPoX (Transaction Processing over XML) [20] and TPCH (Transaction Processing benchmark)[31]. For each benchmark, we used the entity having the most records, namely, the *Orders* document from TPoX and the *LineItems* table from TPCH. We generated datasets of size 800GB each. The *Orders* dataset is more complex than the *LineItems* dataset: *Orders* consists of 181 distinct fields with 4 levels of nesting and varying structure among records, whereas *LineItems* consists of 16 distinct fields with no nesting—i.e., a flat structure—and a rigid schema. These two datasets delineate a range of data scenarios encountered in practice.

4.1 Query Response Time

Plain Hadoop has only one execution plan for applying selection predicates over the data: scanning all splits and applying the selection predicates over each split, i.e., a *full scan*. Therefore, the query response time (or more precisely, the time taken by the filtering step) is mostly independent of the "selectivity" of the query. In contrast, E3 processes only the input splits that are relevant to the query, and hence the query response time depends strongly on the selectivity.

Figures 11 and 12 display the query response time of a conjunctive query of the form `Select count(*) From D Where` P_1 `and` P_2, where `D` is either the *Orders* or *LineItems* dataset, and P_1 and P_2 are equality predicates. The response time is plotted as a function of the query selectivity. This selectivity is expressed

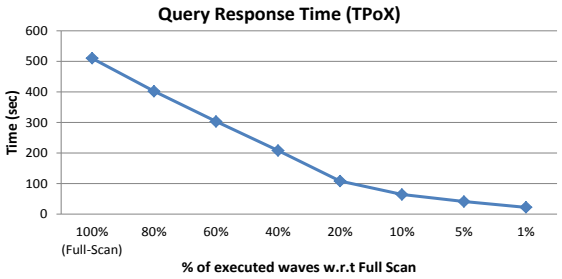


Figure 11: Query response time (TPoX dataset).

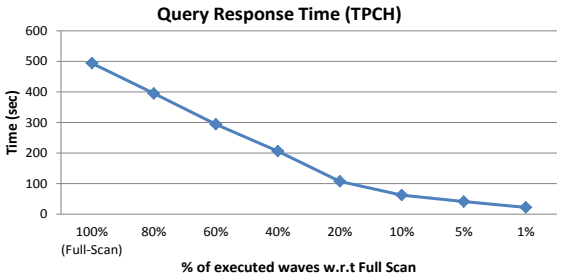


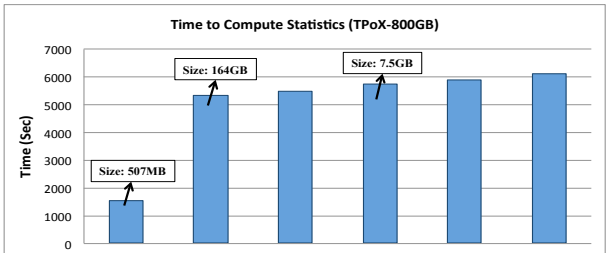
Figure 12: Query response time (TPCH dataset).

as a percentage of the number of waves of mappers that must be executed relative to a full scan, ranging from 100% (full scan) down to 1%. A *wave* of mappers is defined as the maximum number of concurrent mappers that can run in the cluster, i.e., 160 mappers in the iDataPlex cluster. We use this metric because the wave percentage translates directly to the wall-clock response time (assuming the maximum degree of parallelism). As the figures show, by skipping irrelevant splits, and hence reducing the number of executed waves, E3 can achieve speedups in query response time as high as 20x compared to a full scan over the data. For example, if the query selectivity is 1% (or it will hit the materialized view instead of the data file), then the query response time is reduced from 510 seconds (full scan) to 22 seconds. The performance gain is almost the same for both the TPoX and TPCH datasets. Observe that the marginal savings decreases as the selectivity increases beyond 10%, because the initial wave of mappers is very expensive compared to the subsequent waves and hence dominates the cost if the number of waves is small.

4.2 Construction & Storage Costs

We next measured the time required to pre-process the data to construct the various indexes and the materialized view. In these experiments, we fixed the segment limit (k) in range indexing to 20; we study the effect of varying the number of segments in more detail in Section 4.3 below.

Figures 13 and 14 show that building either the inverted index or materialized view is much more expensive than building the range index. To understand this result, first recall that the materialized view is constructed via a map-reduce job followed by a map-only job. The cost of building the inverted index depends mostly on its size. For example, the construction cost for the 164GB TPoX inverted index is roughly 2.5 times higher than for the 41GB TPCH inverted index (compare Figures 13 and 14). The cost of building the range index is much lower because it requires just a single map-only job, and the resulting index size is relatively small. Note that the cost of building several of the indexes is less that the



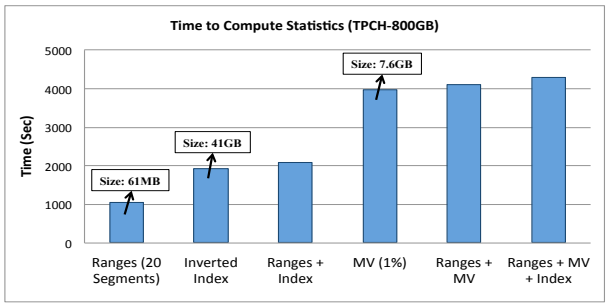


Figure 14: Computation cost (TPCH dataset).

sum of the individual construction costs, because computations are shared during passes over the data; see Section 2.5.

A comparison of Figures 13 and 14 shows that the construction cost of the TPoX indexes is roughly 1.5 times the cost of the TPCH indexes. This disparity aries from the relative complexity of TPoX dataset, which has multiple levels of nesting and many more fields per record (181 in TPoX versus 16 in TPCH). Therefore, around twelve highly selected queries must be executed in order to recoup the construction cost of the TPoX indexes, whereas only eight queries are needed to recoup the cost of the TPCH inverted and range indexes (only four queries are needed if the materialized view is not built).

Figures 13 and 14 also show the storage overheads of the inverted index and the materialized view for both TPoX and TPCH datasets. The materialized view is around 1% of the data size for both datasets, and the inverted index is around 5% of the data size for TPCH and 20% for TPoX. The storage overheads for the range index are discussed in Section 4.3 below.

We note that the MapReduce jobs used to construct the indexes and materialized view were implemented using Jaql scripts. Although this approach allowed rapid development, the use of a general purpose scripting language incurs a performance penalty. Computation times can be improved by using handcrafted MapReduce jobs,

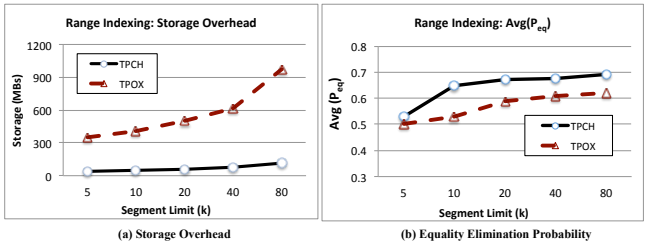


Figure 15: Range Indexing overheads and Elimination Probabilities for various segment limits.

Segment limit(k)	Time (Sec)	P_{eq} [Min, Max, Avg]	P_{range} [Min, Max, Avg]
5	1532	$[4.5 * 10^{-4}, 1.0, 0.5]$	$[6.8 * 10^{-8}, 1.0, 0.48]$
10	1541	$[11 * 10^{-4}, 1.0, 0.53]$	$[16 * 10^{-8}, 1.0, 0.49]$
20	1573	$[25 * 10^{-4}, 1.0, 0.59]$	$[41 * 10^{-8}, 1.0, 0.49]$
40	1606	$[49 * 10^{-4}, 1.0, 0.61]$	$[65 * 10^{-8}, 1.0, 0.50]$
80	1720	$[95 * 10^{-4}, 1.0, 0.63]$	$[118 * 10^{-8}, 1.0, 0.51]$

Table 1: Range index construction times and elimination probabilities (TPoX dataset).

and we are investigating this approach as future work.

4.3 Effect of Segment Limit on Range Index

In this section we focus on the segment limit k used in the domain-segmentation algorithm for constructing the range index, as discussed in Section 2.1. We study the effect of this parameter on the index's storage overhead and elimination effectiveness.

Figure 15(a) shows that as k increases, the total storage overhead also increases, but at a much slower rate. The reason for this slow increase is that the algorithm often uses less than k segments, especially in the case of consecutive integers or dates. This effect is much more pronounced for TPCH dataset than for TPoX, because the TPCH data contains many consecutive dates. The figure depicts absolute storage overheads, which may appear large. The relative overheads, however, are very small; e.g., with 80 segments, the overheads are around 0.1% for TPoX and 0.01% for TPCH.

Recall from Section 2.1 that, in the common scenario where workload information is unavailable, the segments for a given field within a split are chosen to maximize the probability P_{eq} that a uniformly generated equality predicate on the field will cause elimination of the split. This choice also maximizes the elimination probability P_{range} for a uniformly generated range query. If we only keep the (min, max) range for a field within a split, then every selection query that references the field will fall in this range and $P_{\text{eq}} = P_{\text{range}} = 0$.

For the TPoX dataset, Table 1 displays the minimum, maximum and average values of P_{eq} and P_{range} over all of the fields and splits, as a function of the segment limit k. (Results for TPCH are similar.) The minimum values for both P_{eq} and P_{range} correspond to splits for which there are many non-consecutive distinct values for some field, so that it is hard to avoid false positives without using a huge number of segments; conversely, the maximum values correspond to splits in which each field has a small number of distinct values that can be completely contained in k or fewer segments. Figure 15(b) plots the average value of P_{eq} as a function of k for both TPCH and TPoX. Overall, it can be seen that range indexing can yield elimination probabilities of around 50% to 60%, while using only 20 segments, a storage overhead of less than 0.1%. The marginal benefits of increasing k decrease beyond $k = 20$.

4.4 Adaptive Caching

To study the performance of the E3 caching technique, we implemented a simulator that generates synthetic, dynamic query workloads over the TPoX and TPCH datasets. We then used these workloads to compare the SFR and LRU cache replacement policies. As discussed below, we found that SFR outperforms LRU under most query workloads because SFR takes into account the split savings, frequency, and recency of atoms, whereas LRU only considers recency.

Simulation parameters: The simulator has two types of parameters; query workload parameters and cache parameters, as summarized in Figure 16. The *workload size*, i.e., the number of queries in the workload, was set to 100,000 queries in the experiments. The

Workload Parameters

Name	Description
Query workload size	Number of queries in the query workload
Coreset size	Number of value-pairs with high saving if cached
Non-coreset size	Number of value-pairs with low saving if cached
Stickiness window size	Number of distinct value pairs recently queried that will be kept active for a while
Stickiness probability (P_{window})	Probability that the new query is a repetition of a previous query selected from the stickiness window
Coreset probability (P_{core})	Probability that the new query is over a value pair from the coreset

Cache Parameters

Name	Description
Cache size	Maximum number of value pairs that can be cached
Replacement policy	The replacement policy: LRU or SRF

Figure 16: Summary of simulation parameters.

coreset size is the number of nasty atom pairs in the "coreset," which is defined as a set of pairs with very high savings (above 95%) if cached. Similarly, the "non-corset" is defined as a set of atom pairs with low savings if cached, and the *non-coreset size* specifies the number of such pairs. In the experiments, we vary the savings from caching non-coreset items to be below 5%, 30%, or 60%. To model the evolution in popularity (i.e., number of repetitions in the workload) of the different atom pairs, we define a sliding window, called the "stickiness window," over the sequence of queried pairs. The *stickiness window size* determines the number of distinct, recently queried pairs that are likely to be queried again; the larger the stickiness window size, the longer a pair will persist in the workload. A new query in the workload selects randomly and uniformly from the "active" atom pairs in the window with *stickiness probability* P_{window}. The *coreset probability* P_{core} is the probability that the next query, if not a repetition from the stickiness window, will select a value from the coreset items; otherwise, the query selects from the non-coreset items. Thus the workload is generated by flipping a coin with probability P_{window} to either repeat a query from the stickiness window or create a new query. In the latter case, atom pairs are then selected randomly and uniformly from the coreset with probability P_{core} and from the non-coreset with probability $1 - P_{\text{core}}$. The cache parameters consist of the *cache size*, i.e., the number of entries in the cache, and the *replacement policy*, which is either LRU or SFR. In the case of SFR, we experimented with various values of the amplification factor λ, and report results for $\lambda = 1.001$ and $\lambda = 1.0001$.

Evaluation metric: We use the *wave-savings percentage* to measure the effectiveness of the SFR and LRU caching techniques. The wave-savings percentage is an aggregate metric over all queries Q_i in the query workload and is formally defined as follows. Let W_m^i, and W_h^i be the number of waves of mappers executed by query Q_i in the case of a cache miss and a cache hit. The wave-savings percentage is then computed as $1 - (\sum_i W^i_{\text{actual}} / \sum_i W_m^i)$, where W^i_{actual} equals either W_m^i or W_h^i depending on whether Q_i is a cache miss or a cache hit. We use this evaluation metric because it highly correlated with the wall-clock query response time.

Results: The trellis plot in Figure 17 displays the results of a set of experiments in which the distinction between the coreset and non-coreset items is very sharp. That is, the savings from caching a non-coreset item is below 5% compared to 95% savings from caching a coreset item. In this case, the coreset items are "precious" and need to be detected and cached. In the experiment, we use coreset and non-coreset sizes of either 1,000 or 10,000 atom pairs, and the stickiness-window sizes of either 10 or 1,000 active atom pairs. We set P_{window} to 0.1, 0.3, and 0.6 and P_{core} to 0.2, 0.5, and 0.8. The cache size is set to 1,000. We observed that *non-coreset size* and *stickiness window size* do not discriminate between LRU and SFR, and hence are omitted from the figure for clarity.

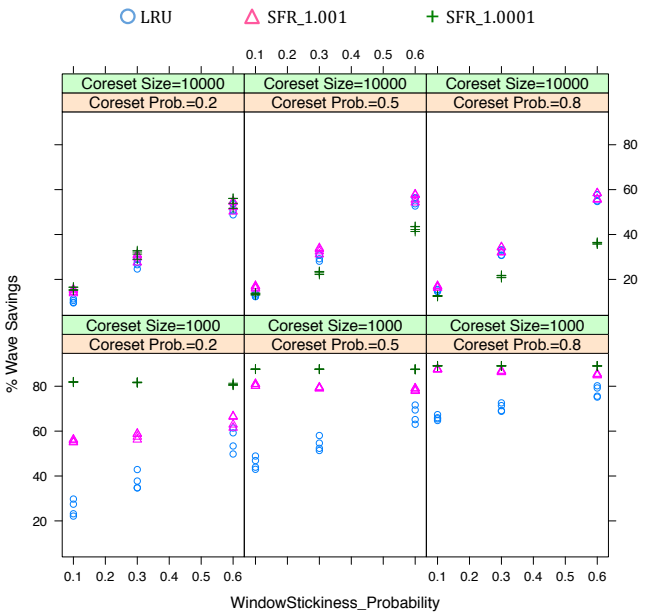


Figure 17: Cache performance (5% Savings from Non-coreset)

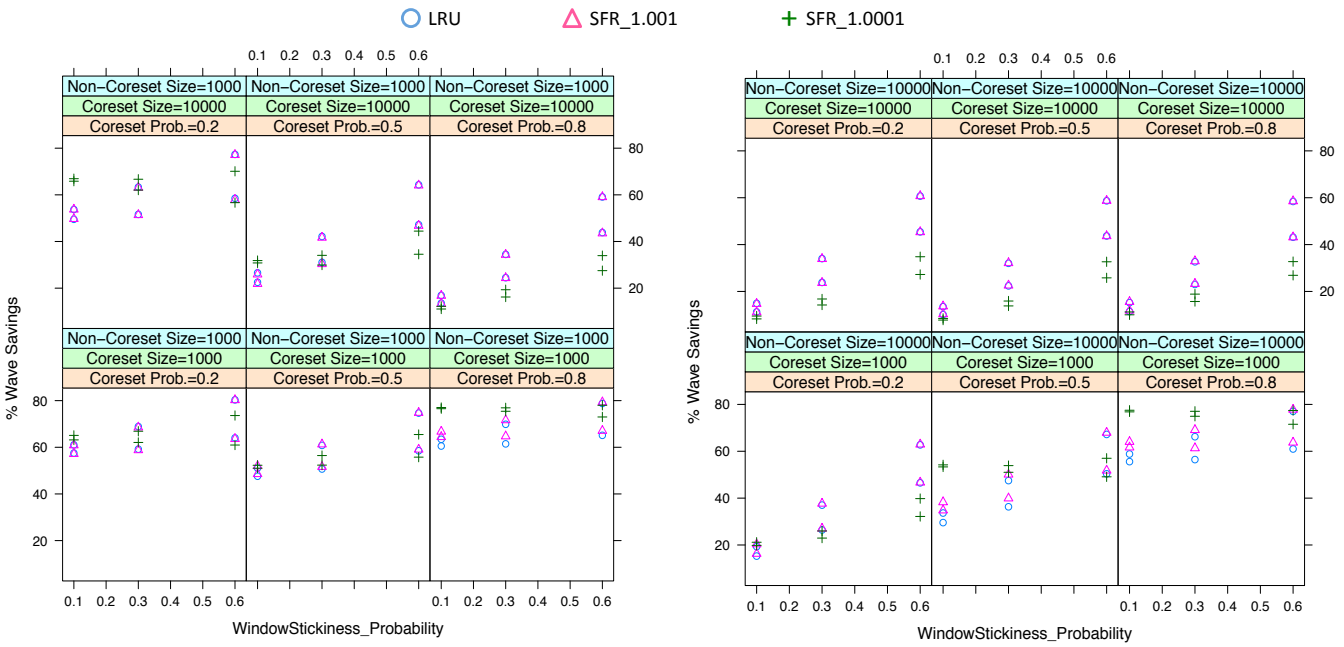
Figure 18 displays results in a scenario where the difference between coreset and non-corset items is muted. Specifically, non-coreset items have up to 60% savings in splits if cached. We observed that *non-coreset size* becomes an important factor in cache performance in these experiments, so we present results separately in Figures 18(a) (non-coreset size $= 1,000$) and 18(b) (non-coreset size $= 10,000$).

The two key findings are as follows.

1. SFR performs consistently better than LRU, especially when the amplification factor λ is set to 1.001. The reason is that SFR is able to detect and cache valuable coreset items whereas LRU gets distracted by the non-coreset items.

2. SFR with $\lambda = 1.001$ performs well in virtually all experimental conditions, and is the best or near-best of the methods in the majority of situations. In additional experiments with other values of λ, not reported here, the value of 1.001 was found to be the most robust. The exception is when coreset items are precious and the coreset can fit into the cache (lower half of Figure 17). In this case, use of the smaller value of $\lambda = 1.0001$ puts greater emphasis on savings, so that the coreset, once in the cache, stays in the cache, thereby maximizing performance. (Even then, SFR with $\lambda = 1.001$ is almost the best algorithm in most cases.)

In our final experiment, we study the resilience of SFR to sudden changes in the query workload by instantaneously replacing the entire coreset by a completely different coreset halfway through the workload. LRU is expected to adapt quickly to this kind of "shock" since it only considers recency when caching items. In the experiment, we set the coreset and non-coreset sizes to 1,000 and the parameters P_{core} and P_{window} to 0.8 and 0.1. Notice that because P_{core} is high, the cache performance is sensitive to the coreset items, and because P_{window} is small, the effect of changing the coreset items is magnified. We set the savings from caching a non-coreset item to below 5%. To compare the behavior of LRU and SFR, we divided the workload (100,000 queries) into small non-overlapping windows



(a) Non-Coreset size = 1000

(b) Non-Coreset size = 10,000

Figure 18: Cache performance (60% Savings from Non-coreset).

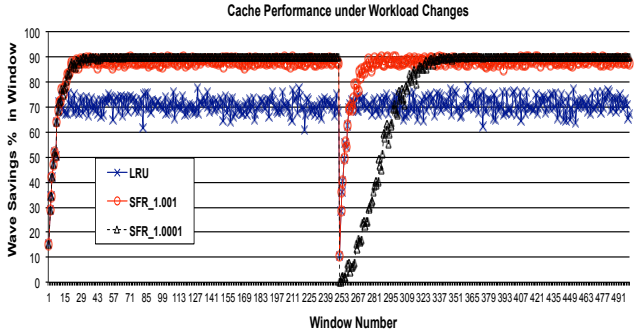


Figure 19: Cache performance under a workload change.

of size 200 each, and then measured the wave-savings percentage over each window separately

Figure 19 displays the results. The x-axis in the figure shows the sequential window number (from 1 to 500) and the y-axis shows the average savings in each window. As can be seen, SFR with $\lambda = 1.001$ adapts as fast as LRU while yielding greater savings that are 20% larger. In contrast, when $\lambda = 1.0001$, SFR adapts more slowly to the changes since it puts more weight on savings than on recency. The results show also that SFR, in general, does not require too many queries to start outperforming LRU. For example, SFR started to outperform LRU after around 10 windows.

5. RELATED WORK

Despite the desirable features of Hadoop—such as scalability, flexibility, low start-up costs, and attractive price-performance characteristics—a performance gap remains between Hadoop and parallel databases; see, e.g., [8, 27].

The Hadoop++ system [8] incorporates "Trojan index" and "Trojan join" techniques to enhance the performance of selection and join queries, respectively. These techniques, however, require changes to the data layout, specifically, augmenting the data with "Trojan files." Jiang et al. [16] propose several techniques to enhance Hadoop's performance, such as using range-indexes (building range statistics for each chunk of data on a sorted field), block-level indexes (building an index for each chunk of data over an unsorted field), and other I/O and memory-based optimizations. Unlike E3, all of the foregoing methods initiate map tasks for each split in the input file even when a split is irrelevant to the query. Moreover, these techniques require physical design: users need to specify the data that is to be indexed. Finally, Hadoop++ does not handle the cases addressed by E3's materialized view and adaptive caching mechanisms. As discussed previously, a recent extension of Hadoop++ called HAIL [9] has demonstrated that record-level indexes do not yield wall-clock savings unless HDFS and Hadoop are thoroughly re-engineered. In contrast, our techniques can immediately be applied to HDFS and Hadoop as-is.

HadoopDB [1] proposes heavyweight changes to the Hadoop framework by storing the data in a local DBMS. HadoopDB thereby enjoys the benefits of a DBMS such as query optimization and use of indexes. These benefits, however, come at the cost of disrupting the dynamic scheduling and fault tolerance mechanisms of Hadoop; the data is no longer under the control of HDFS but is managed by the DBMS, and so cheap commodity hardware no longer suffices.

To optimize regular-expression queries over text fields, Lin et al. [17] propose building a full-text inverted index at the level of the blocks used by the Fedora DBMS data compression module (LZO). In contrast, E3 supports selection queries over any data type, e.g., strings, numbers, dates, and it also handles range predicates. Moreover, our proposed techniques go beyond use of inverted indexes to handle cases where the index does not help, e.g., building a materialized view over infrequent, scattered values and maintaining a cache of value pairs that are highly selective.

Several techniques have been proposed for optimizing join and ag-

gregation queries in Hadoop. These include co-partitioning the data for fast joins [8], co-locating files that are expected to be joined or aggregated together [10], supporting arbitrary theta-join in Hadoop where the join condition is not limited to equality predicates [21], supporting set-similarity joins [32], and optimizing join operations in generic MapReduce environments [4, 2, 18]. These methods are orthogonal to the techniques proposed in this paper as they focus mostly on join and aggregation queries and do not address optimization of selection queries.

The literature contains several instances of caching strategies related to ours; see [25] for a survey. For traditional DBMSs, Palpanas et al. [24] provide an adaptive strategy for caching intermediate and final query results to speed up the execution of subsequent queries. The proposed replacement policy takes into account the expected execution-time savings from a cached object, as well as the frequency of accesses to the object. In [26], exponential damping is used in the context of web caching as a method for reducing the importance of prior queries that have not been executed recently. Our adaptive caching method is inspired by these techniques and uses a novel cache replacement policy that takes into account savings, frequency and recency, and is adapted to a Hadoop environment.

Partition elimination (also known as data localization [23]) is a common technique used by parallel database systems. E3 differs from such systems by using not only partitioning attributes, but all data fields for split elimination, and by not requiring an a priori physical database design or a given query workload for selecting indexes and materialized views. Our use of range statistics is in the spirit of [19], which proposed storing aggregates such as min and max for blocks of contiguous tuples in a relational data warehouse and exploiting these values for selection predicates and aggregation queries. We use this idea in a different context, with random access to the range statistics and simple, explicit algorithms for segmentation.

6. CONCLUSION

We have presented the E3 indexing framework for speeding up queries with selection predicates in a Hadoop environment. E3 is potentially applicable to an important class of analytical settings because it does not require data movement, a particular physical data layout, or prior knowledge about the query workload. Instead, E3 employs a novel combination of split-level range index and inverted indexes, a special materialized view, and an adaptive cache to avoid processing irrelevant data splits. E3's split-elimination techniques avoid unnecessary I/O and mapper-startup costs. Our experiments on both semi-structured (TPoX) and structured (TPCH) datasets show that, while incurring only a modest storage overhead, use of E3 can reduce query response times by up to a factor of 20. E3 exploits the fact that many attributes—beyond just partitioning key(s)—are effective in split elimination. This inclusive approach significantly differentiates E3 from both parallel databases and from earlier attempts at query optimization in Hadoop [1, 8, 16].

7. REFERENCES

[1] A. Abouzeid et al. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *PVLDB*, 2(1):922–933, 2009.

[2] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, 2010.

[3] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C.-C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011. `http://code.google.com/p/jaql`.

[4] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD*, pages 975–986, 2010.

[5] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[6] G. B. Dantzig. Discrete-variable extremum problems. *Oper. Res.*, 5(2):266–288, 1957.

[7] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010.

[9] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *PVLDB*, 5(11):1591–1602, 2012.

[10] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. CoHadoop: Flexible data placement and its exploitation in Hadoop. *PVLDB*, 4(9):575–585, 2011.

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

[12] S. W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theor.*, 10(3):399–401, 1966.

[13] The Apache Hadoop Project. `http://hadoop.apache.org/core/`, 2009.

[14] D. Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. Technical Report GIT-CERCS-09-1, Georgia Institute of Technology, 2009.

[15] O. Ibarra and C. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463–468, 1975.

[16] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[17] J. Lin, D. Ryaboy, and K. Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In *MapReduce*, pages 59–66, 2011.

[18] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: leveraging columnar storage for scalable join processing in the MapReduce framework. In *SIGMOD*, pages 961–972, 2011.

[19] G. Moerkette. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.

[20] M. Nicola, I. Kogan, and B. Schiefer. An XML transaction processing benchmark. In *SIGMOD*, pages 937–948, 2007.

[21] A. Okcan and M. Riedewald. Processing theta-joins using MapReduce. In *SIGMOD*, pages 949–960, 2011.

[22] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*, pages 1099–1110, 2008.

[23] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2011. Third edition.

[24] T. Palpanas, P. Larson, and J. Goldstein. Cache management policies for semantic caching. Technical Report CSRG-439, Dept. of Computer Science, University of Toronto, 2001.

[25] S. Podlipnig and L. Pöszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374 – 398, 2003.

[26] M. Reddy and G. P. Fletcher. Exp1: a comparison between a simple adaptive caching agent using document life histories and existing cache techniques. *Computer Networks and ISDN Systems*, 30(22-23):2149–2153, 1998.

[27] M. Stonebraker et al. MapReduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[28] Z. Svitkina and L. Fleischer. Submodular approximation: Sampling-based algorithms and lower bounds. *SIAM J. Comput.*, 40(6):1715–1737, 2011.

[29] The Apache Software Foundation. HDFS architecture guide. `http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html`.

[30] A. Thusoo et al. Hive - a warehousing solution over a Map-Reduce framework. *PVLDB*, 2(2):1626–1629, 2009. `https://cwiki.apache.org/Hive/tutorial.html`.

[31] TPC-H specification 2.8.0. `http://www.tpc.org/tpch`.

[32] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *SIGMOD*, pages 495–506, 2010.