Efficient and Flexible Index Access in MapReduce

Zhao Cao
IBM Research
Beijing, China
caozhao@cn.ibm.com

Shimin Chen*
State Key Laboratory of Computer Architecture
Institute of Computing Technology
Chinese Academy of Sciences
chensm@ict.ac.cn

Dongzhe Ma
Tsinghua University
Beijing, China
mdzfirst@yahoo.com.cn

Jianhua Feng
Tsinghua University
Beijing, China
fengjh@tsinghua.edu.cn

Min Wang
Google Research
Mountain View, CA, USA
minwang@google.com

ABSTRACT

A popular programming paradigm in the cloud, MapReduce is extensively considered and used for "big data" analysis. Unfortunately, a great many "big data" applications require capabilities beyond those originally intended by MapReduce, often burdening developers to write unnatural non-obvious MapReduce programs so as to twist the underlying system to meet the requirements. In this paper, we focus on a class of "big data" applications that in addition to MapReduce's main data source, require selective access to one or many data sources, e.g., various kinds of indices, knowledge bases, external cloud services.

We propose to extend MapReduce with *EFind*, an *E*fficient and *F*lexible *ind*ex access solution, to better support this class of applications. EFind introduces a standard index access interface to MapReduce so that (i) developers can easily and flexibly express index access operations without unnatural code, and (ii) the EFind enhanced MapReduce system can automatically optimize the index access operations. We propose and analyze a number of index access strategies that utilize caching, re-partitioning, and index locality to reduce redundant index accesses. EFind collects index statistics and performs cost-based adaptive optimization to improve index access efficiency. Our experimental results, using both real-world and synthetic data sets, show that EFind chooses execution plans that are optimal or close to optimal, and achieves a factor of 2x–8x improvements compared to an approach that accesses indices without optimization.

1. INTRODUCTION

MapReduce [6] is one of the most popular programming paradigms in the cloud. As the amount of digital information is exploding [11], MapReduce is extensively considered and used for processing the so-called "big data", such as web contents, social media, click streams, event logs, and so on. Higher-level query languages, such as Pig [15] and Hive [19], facilitate data analysis with friendly SQL-like interface on top of MapReduce. MapReduce programming is still important because these higher-level languages mainly provide data processing functionality when data schemas are declared, but essentially fall back to MapReduce programming for unstructured and semi-structured data.

*Corresponding author

(c) 2014, Copyright is with the authors. Published in Proc. 17th International Conference on Extending Database Technology (EDBT), March 24-28, 2014, Athens, Greece: ISBN 978-3-89318065-3, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

In this paper, we are interested in improving MapReduce to better support a class of "big data" applications that require *selective access* to *one or many* data sources in addition to MapReduce's main input data source. MapReduce is originally designed to sequentially scan and process a single input data source for a job. As a result, developers often have to write unnatural non-obvious MapReduce programs so as to twist the underlying MapReduce system to selectively access extra data sources. We believe that a good solution to this problem benefits MapReduce programming as well as higher-level query languages on top of MapReduce.

Big Data Applications Requiring Index Access. We use the word "index" broadly in this paper to mean data sources that allow selective accesses, including but not limited to database-like indices, inverted indices, key-value stores, knowledge bases, and data sources behind cloud services. We use the term "index access" to mean selective access to data sources that are not the main input data in MapReduce. From our experience, index accesses are often required in the following applications:

- *Text analysis.* Unstructured text analysis is an important task for analyzing web contents and social media. Text analysis often requires accessing indices, e.g., inverted indices [23], precomputed acronym dictionaries [8], and knowledge bases such as Wikipedia [13].
- *Index-based joins.* Present join implementations on MapReduce are mainly scan based. Index-based joins, such as index nested loop join and join using bitmap indices, have been shown to out-perform scan-based joins under high join selectivity or for analyzing read mostly data [16, 10]. Hence, the capability to naturally express index-based joins on MapReduce is desirable.
- *Location-based analysis.* Location-based analysis analyzes location information to model user preferences [14] and to categorize location types [21], in order to provide more personalized services and better quality recommendations. An example algorithm is k-nearest neighbor join between two spatial data sets [22]. Spatial indices are often required in such analysis.
- *Data sources behind cloud services.* The utility computing model of the cloud lowers the cost for developing cloud services. When an organization does not own all the data for supporting her analysis needs, she can subscribe to third-party cloud services to obtain relevant data. We consider a cloud service as a selec-

tively accessed index because a user is often charged on a pay-per-use basis. Hence we would like to reduce accesses to such cloud service as much as possible. It is preferable to flexibly and seamlessly integrate cloud services into MapReduce.

Interestingly, indices in the above examples can be dynamic in that given a search key the return value is dynamically computed. For example, a knowledge base index can use machine learning classifiers to compute topics from input text. (More details will be shown in Figure 4.) Note that this index can compute results for any input text, thus the number of valid keys (valid input texts) is infinite, which is very different from traditional indices. While accessing extra data sources using traditional indices may be regarded as index-based joins, such dynamic computation-based index access cannot be supported by traditional join operations, such as index-based, hash-based, or sort-based joins.

Accessing Index is a Pain in Vanilla MapReduce. Unfortunately, vanilla MapReduce lacks efficient and flexible support for selectively accessing data sources that are not the main input data [17, 18]. To get around this limitation, current solution is to hand-code index access in Map or Reduce functions, which are treated as black boxes by the runtime MapReduce system. However, there are two problems of this approach. First, the MapReduce system has no way to know about the index operations, let alone optimizing them. To make matters worse, index access often performs network and disk operations, interfering with the normal I/O operations in MapReduce. Second, the burden of achieving good index access performance is entirely on developers. A developer has to spend a lot of time and effort to analyze the index access characteristics and fine tune the index operations, considering issues such as data transfer and distributed computing. This essentially violates MapReduce's design principle of hiding data transfer and distribution details from developers. The resulting code could be error-prone and difficult to reuse for other index access tasks.

Our Solution: EFind. We propose *EFind*, an Efficient and Flexible index access solution in MapReduce. As illustrated in Figure 1, EFind is a *connection layer* between MapReduce and indices. Note that EFind does NOT implement any indices by itself. From EFind's viewpoint, an index is a black box. The user-provided lookup method implements the actual index access functionality, as will be described in Section 2. We aim to support efficient index access while minimizing restriction on both MapReduce and indices:

- *Index Flexibility*: We propose a flexible index access interface for easily incorporating index access into MapReduce. There are four dimensions of index flexibility: *(1) What type of index is used?* EFind interface can be easily programmed to support different types of indices, e.g., database-like indices, inverted indices, key-value stores, and so on. *(2) Where is an index invoked?* Index access can be configured before Map, in between Map and Reduce, and after Reduce in a MapReduce data flow. *(3) How is an index invoked?* Developers can easily customize the invocation of an index by optionally providing pre-processing and post-processing procedures for accessing an index, e.g., to implement filtering or data field projection. *(4) How many indices are used?* EFind is capable of expressing and supporting multiple indices to be used in a single MapReduce job. Our only assumption is that an index lookup with the same key returns the same result during an EFind enhanced job. There is no other restriction on indices. Indexed data sources can be structured, semi-structured, or unstructured. Indices can be tree-based or hash-based, or even dynamic.
- *MapReduce Flexibility*: For MapReduce, we do not place any restriction on the functionality of Map/Reduce functions and the structure of main input data from HDFS.
- *Efficiency*: For an index access operation expressed using the EFind interface, the EFind enhanced MapReduce system automatically optimizes the operation without user intervention. We study a number of index access strategies that aim to reduce the redundancy of index accesses. EFind performs cost-based optimization to choose the best index access strategy. It collects index access statistics on the fly, and adaptively re-optimizes a job when necessary. In this way, EFind achieves efficient index access, while freeing application developers from the difficult task of fine tuning index operations. Like the MapReduce design, EFind hides the details of distributed computation for simplicity of programming.

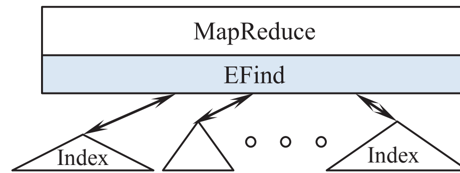


Figure 1: EFind provides an efficient and flexible index access connection layer between MapReduce and indices.

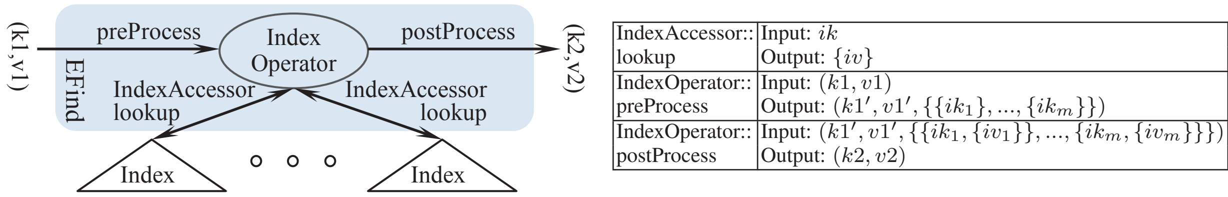
Related Work. Our work is in the same spirit as recent studies that improve various aspects of MapReduce [1, 12, 7]. To the best of our knowledge, our work is the first to propose a flexible and efficient solution for incorporating index access into MapReduce.

Indices have been extensively studied in relational database systems. An RDBMS has full knowledge of relational operations and index structures. Therefore, it can transparently make decisions on whether to use indices for a query. In contrast, the MapReduce setting is different and challenging because (i) the extensive use of user-defined functions (e.g., Map and Reduce) makes it difficult to understand the semantics and use patterns of index access, and (ii) a wide variety of index types and index invocation behaviors need to be supported. Consequently, it is difficult to transparently use an index as in RDBMS. Instead, users have to manually place index operators into a MapReduce data flow. Compared to a naïve solution, EFind significantly reduces user efforts and automatically optimizes index access operations while maintaining the flexibility of the MapReduce programming and index accesses.

Higher-level query languages such as Pig [15] and Hive [19] provide friendly SQL-like interface on top of MapReduce. However, their optimization is limited when user-defined functions exist or data schema is unclear. In constrast, EFind does not have such limitation. EFind supports arbitrary user-defined functions and even external cloud services. We believe that these higher-level query languages can employ EFind to achieve flexible index access.

There are a large number of studies on designing efficient indices or distributed data services [2, 5, 4]. Note that EFind is not a distributed index in itself. EFind provides the glue logic that incorporates index access into MapReduce.

Contributions. This paper makes the following contributions. First, we propose the EFind index access interface. To our knowledge, this is the first paper that introduces a flexible index access interface to MapReduce to support various index types, various index access locations in a MapReduce data flow, flexible customization of index invocation behaviors, and multiple indices in a single MapReduce job. Second, we propose and analyze a number of index access strategies, including caching, re-partitioning, and index locality optimizations. Third, we design and implement an adaptive optimization framework for EFind that is capable of collecting index statistics on the fly and dynamically choosing the best index access



IndexAccessor:: lookup	Input: ik Output: $\{iv\}$
IndexOperator:: preProcess	Input: $(k1, v1)$ Output: $(k1', v1', \{\{ik_1\}, ..., \{ik_m\}\})$
IndexOperator:: postProcess	Input: $(k1', v1', \{\{ik_1, \{iv_1\}\}, ..., \{ik_m, \{iv_m\}\}\})$ Output: $(k2, v2)$

Figure 2: An EFind IndexOperator class accesses one or multiple indices at a single point in the MapReduce data flow. The `IndexAccessor::lookup` **method(s) is implemented once for each type of index, while the** `IndexOperator::preProcess` **and** `IndexOperator::postProcess` **methods customize index access behaviors at a specific point in a specific MapReduce job.**

```
public class UserProfileAccessor extends IndexAccessor{
      String server; int port;
      public UserProfileAccessor(String server_desc) {
            // extract server and port from server_desc, then create socket connection to server:port
      }
      public Vector<Writable> lookup(Writable ik) {
            // look up ik in the index and return results
      }
}
public class UserProfileIndexOperator extends IndexOperator{
      public void preProcess(Writable k1, Writable v1, IndexInput iklist) {
            String user = extractUserAccount(v1);
            iklist.put(1, user); // {{ik}}
            v1 = removeOtherFields(v1); // v1'
      }
      public void postProcess(Writable k1, Writable v1, IndexOutput indexValues,
                              OutputCollector<Writable, Writable> output){
            String profile = indexValues.get(0).getAll()[0];
            String city = extractCity(profile);
            String result = ((Text) v1).toString().append(city);
            output.collect(k1, result);
      }
}
```

Figure 3: An implementation of an IndexAccessor and an IndexOperator that obtain the city information from a user profile index for a tweet in Example 2.1.

strategies. Finally, we present an extensive experimental study using both real-world and synthetic data sets. Experimental results show that EFind selects execution plans that are optimal or close to optimal, and achieves a factor of 2x–8x improvements compared to an approach that accesses indices without the optimization.

Outline. The rest of the paper is organized as follows. Section 2 proposes EFind's index access interface. Then Section 3 presents and analyzes a number of index access strategies, followed by the description of EFind's cost-based adaptive optimization in Section 4. Section 5 empirically evaluates the proposed EFind solution. Finally, Section 6 concludes the paper.

2. EFIND INDEX ACCESS INTERFACE

In this section, we propose an EFind programming interface for easily and flexibly integrating indices into MapReduce. For concrete presentation, we show code that extends Hadoop [3], a popular open-source implementation of MapReduce, while the proposal is applicable to any MapReduce implementation. We use the following example to illustrate the interface.

EXAMPLE 2.1. *An analyst would like to understand the spatio-temporal patterns of popular topics from a large number of Twitter messages collected over a period of several months. She would like to compute the top-k most popular Twitter topics for every combination of (city, day), then associate the topics with important news events. Every tweet contains a twitter user account, a timestamp, a message, and a few other fields. The computation takes five steps:*

1) *Look up the user account of every tweet in a* user profile index *to obtain the cities for the tweets;*
2) *Extract keywords from every tweet message;*
3) *Call an external* knowledge-base service *to convert the keywords of every tweet into a topic;*
4) *Group tweets according to combinations of (city, day), then compute the top-k popular topics for each group;*
5) *For each combination of (city, day), use an* event database *to enrich the result with important local, national, and global news events.*

Note that Step 1), 3), and 5) need to access three indices, respectively. The index in Step 3) uses machine learning classifiers to dynamically compute a topic for a given input tweet. It would take a lot of manual coding and tuning efforts to effectively use the three indices in the original MapReduce. In the following, we show how to easily express the computation with EFind.

EFind Programming Interface. Figure 2 depicts the EFind interface that accesses one or multiple indices at a single place in a MapReduce job. Following the convention in MapReduce, the input is a key-value pair (k1, v1), and the output result of the index access is also a key-value pair (k2, v2). The interface specifies three user-defined methods in two separate classes, i.e., $lookup$ in $IndexAccessor$, and $preProcess$ and $postProcess$ methods in $IndexOperator$. The $IndexAccessor$ class is implemented for each type of index and can be reused for the same type of index, while $IndexOperator$ contains invocation specific code that customizes the index behaviors for a specific MapReduce job.

The input and output parameters of the three methods are listed in the table in Figure 2. $lookup$ takes an index key ik as input

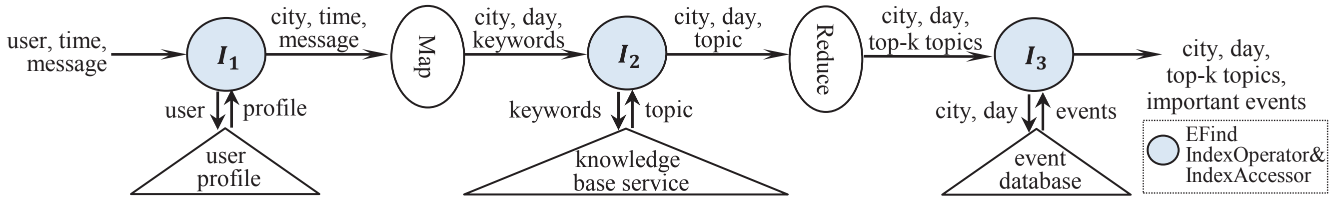


Figure 4: An EFind-enhanced MapReduce job that computes spatio-temporal topic patterns from tweets for Example 2.1. Three EFind IndexOperators are placed before Map, in between Map and Reduce, and after Reduce, respectively. The knowledge base service dynamically computes topics using machine learning classifiers, while the other two indices perform traditional lookups.

```
public class JobDriver{
  public void run(){
    IndexJobConf iConf = new IndexJobConf();
      // 1. look up user profile before Map
    UserProfileIndexOperator I1 = new UserProfileIndexOperator();
    I1.addIndex("indexaccessor.UserProfileAccessor","cassandra,localhost,9160,userprofile");
    iConf.addHeadIndexOperator(I1);
      // 2. Map
    iConf.setMapper(KeyWordExtractMapper.class);
      // 3. obtain topic between Map and Reduce
    TopicCategoryIndexOperator I2 = new TopicCategoryIndexOperator();
    I2.addIndex("indexaccessor.TopicCategoryAccessor","external.TopicCategoryService");
    iConf.addBodyIndexOperator(I2);
      // 4. Reduce
    iConf.setReducer(TimeRangeCityGroupReducer.class);
      // 5. look up important events after Reduce
    ImportantEventIndexOperator I3 = new ImportantEventIndexOperator();
    I3.addIndex("indexaccessor.ImportantEventAccessor","mySQL,15.154.147.160,3216,event");
    iConf.addTailIndexOperator(I3);
    iConf.submit();
  }
}
```

Figure 5: A MapReduce job driver that configures the three EFind IndexAccessors and IndexOperators in Figure 4 for Example 2.1.

and returns a list of results $\{iv\}$. $preProcess$ takes the input key-value pair $(k1, v1)$, extracts one key list $\{ik_j\}$ for every index j to access ($j = 1, 2, ...m$), and optionally modifies $(k1, v1)$. $postProcess$ combines index lookup results into the result key-value pair $(k2, v2)$, while performing optional operations, such as applying filtering criteria.

Figure 3 shows an example implementation for accessing the user profile index in Example 2.1. The $UserProfileAccessor$ is implemented as a sub-class of $IndexAccessor$. This class can be reused whenever a user profile index is to be accessed. The constructor method records the server and port information for accessing the index, then creates a socket connection to the server. The $lookup$ method will send and receive messages using the socket connection to access the user profile index. Moreover, we implement a $UserProfileIndexOperator$ class as a sub-class of $IndexOperator$. The $preProcess$ and $postProcess$ methods deal with the tasks of extracting the user account from a tweet and extracting the city information from the index lookup result, which are specific to support Example 2.1.

Note that in general, an index can be distributed across a large number of machine nodes. The server in the constructor of an $IndexAccessor$ class specifies an entry point to the index, e.g., the root node in a distributed B-tree [2], the central metadata server in a master-worker organized index (e.g., BigTable [5]), or one of the index servers in a p2p organized index (e.g., Cassandra [4]). In each case, we can further obtain a list of index servers for optimization purpose, which we will discuss in Section 3.

Placing EFind Index Operators in a MapReduce Job. Our design follows the convention of MapReduce to take (generate) key-value pairs as inputs (outputs) of an $IndexOperator$. In this way, one can easily place an $IndexOperator$ at any location in a MapReduce data flow, i.e., before Map, between Map and Reduce, and after Reduce. In other words, the input of an $IndexOperator$ can come from the main input of MapReduce, the output of Map, or the output of Reduce. The output of an $IndexOperator$ can provide input to Map, input to Reduce, or the final output of MapReduce. Moreover, several $IndexOperator$s can be linked next to each other in a MapReduce data flow.

Figure 4 shows the high-level picture of an EFind based implementation of Example 2.1. Step 1)–5) of Example 2.1 are implemented with I_1, Map, I_2, Reduce, and I_3, respectively. The user profile index, knowledge base, and event database are accessed at I_1 before Map, at I_2 between Map and Reduce, and I_3 after Reduce, respectively. Map mainly extracts keywords from tweet messages, while the group-by and top-k computation fits well with the computation model of Reduce.

This computation can be easily configured in a MapReduce job driver, as shown in Figure 5. A vanilla MapReduce job is specified with a $JobConf$ class. We extend it into an $IndexJobConf$ class to express an EFind enhanced MapReduce job. Specifically, it supports three additional methods, $addHeadIndexOperator$, $addBodyIndexOperator$, and $addTailIndexOperator$, that insert EFind $IndexOperator$s before Map, between Map and Reduce, and after Reduce, respectively. Figure 5 shows how the three $IndexOperator$s are configured for Example 2.1. Note that the $addIndex$ method specifies the $IndexAccessor$ class and its constructor parameter (e.g., server description) for an $IndexOperator$.

Achieving Four Dimensions of Index Flexibility. (1) "What": A new type of index can be supported by implementing a new $IndexAccessor$ class; (2) "Where": EFind index access can be

easily configured at any point in a MapReduce data flow by specifying an $IndexJobConf$ class; (3) "How": The $IndexOperator$ class can customize index access behaviors for a specific MapReduce job; and (4) "How many": multiple indices can be accessed at any location in a MapReduce data flow either by configuring multiple $IndexAccessor$ classes for a single $IndexOperator$ or by linking multiple $IndexOperator$s together. The index accesses are independent in the former, but may be dependent in the latter.

3. INDEX ACCESS STRATEGIES AND COST ANALYSIS

The EFind programming interface exposes index access operations to the EFind enhanced MapReduce system. The system can automatically optimize index operations to achieve good performance. In this section, we propose and analyze a number of execution strategies for accessing indices.

Note that unlike database join optimizations, where various join orders result in the same correct results, it is not an option to reorder $IndexOperator$s for two reasons. First, the placement of an $IndexOperator$ in a MapReduce data flow is often restricted by the data dependencies in the computation. For example, in Figure 4, accessing knowledge base at I_2 depends on the keywords extracted in the Map step. Second, user defined functions in MapReduce make it difficult to understand the computation logic. Without such understanding, it would be unsafe to reorder $IndexOperator$s. Consequently, we consider each $IndexOperator$ instance separately when optimizing index access. On the other hand, EFind does provide a feature to specify independent index accesses: a single $IndexOperator$ can access multiple independent indices. Then, the system will compute an optimal plan for accessing the multiple indices. Application developers are advised to take advantage of this feature to give EFind more freedom for performing optimizations.

In the following, we consider four index access strategies for the case where an $IndexOperator$ is associated with one index in Section 3.1 to 3.4. Then we discuss the more sophisticated case where multiple indices are associated with an $IndexOperator$ in Section 3.5. Terms used in the analysis are summarized in Table 1.

3.1 Baseline Strategy

Figure 6 shows the baseline strategy for implementing an EFind $IndexOperator$ that accesses a single index. Specifically, we take advantage of the chained function feature in existing MapReduce systems (e.g., Hadoop). In a chain of functions, the output of one function is the input to the next function on the chain. Both the Map computation and the Reduce computation can take a chain of user defined functions.

From left to right in Figure 6, we show the cases where the $IndexOperator$ is before Map, between Map and Reduce, and after Reduce, respectively:

- *Before Map:* $preProcess$, $lookup$, and $postProcess$ are inserted as three chained functions before the original Map function as part of the Map computation.
- *Between Map and Reduce:* The methods $preProcess$, $lookup$, and $postProcess$ are inserted as three chained functions after the original Map function as part of the Map computation.
- *After Reduce:* $preProcess$, $lookup$, and $postProcess$ are inserted as three chained functions after the original Reduce function as part of the Reduce computation.

For the second case, we choose to implement the three methods as part of the Map computation rather than part of the Reduce computation. This gives application developers the most flexibility in choosing group-by keys, who can either choose the Map output key as the group-by key by setting $k2 = k1$ in the $IndexOperator$ or compute a different group-by key in $postProcess$.

Table 1: Terms used in cost analysis.

Term	Description
$N1$	Average number of input $(k1, v1)$ to $preProcess$ on a single machine
$S1$	Average size of input $(k1, v1)$ to $preProcess$
Nik_j	Average number of index lookup keys per input $(k1, v1)$ for index j
Sik_j	Average size of index keys for index j
Siv_j	Average size of lookup results per lookup key for index j
T_j	Average time for index j to serve a lookup
BW	Network bandwidth between two machines in the computation environment
T_{cache}	Average time for a probe in the lookup cache
R	Miss ratio of the lookup cache
$Spre$	Average size of output $(k1', v1', \{\{ik_1\}, ..., \{ik_m\}\})$ of $preProcess$ per $(k1, v1)$ input
$Sidx$	Average size of output $(k1', v1', \{\{ik_1, \{iv_1\}\}, ..., \{ik_m, \{iv_m\}\}\})$ of $lookup$ per $(k1, v1)$ input
$Spost$	Average size of output of $postProcess$ per $(k1, v1)$ input
$Smap$	Average size of output of the original Map per input
f	Average cost of storing and retrieving a byte from the distributed file system
Θ	Average number of duplicates per index lookup key

The cost of the baseline strategy is computed as the sum of the costs of $preProcess$, $lookup$, and $postProcess$, respectively. As all the index access strategies pay similar local computation costs for $preProcess$ and $postProcess$, we can omit them in the cost analysis formulae without changing the relative costs of different index access strategies. Therefore, we focus on computing the cost of $lookup$ for index j:

$$Cost_{base} = N1 \cdot Nik_j(\frac{Sik_j + Siv_j}{BW} + T_j) \tag{1}$$

$N1 \cdot Nik_j$ computes the total number of index keys that a single machine searches in the index. The cost of looking up a key has two parts: index local computation time (T_j) and network transfer cost ($(Sik_j + Siv_j)/BW$)[1].

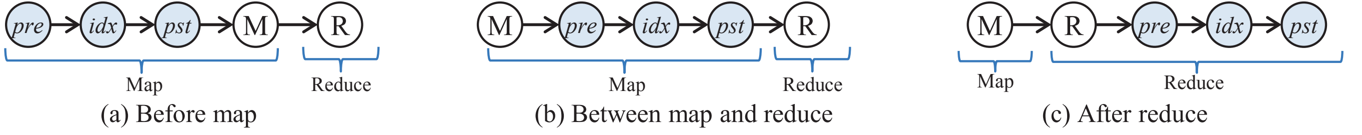
3.2 Lookup Cache Strategy

We observe that index lookups are typically read-only. Therefore, we assume that an index lookup is an idempotent operation. That is, the index data stays unchanged (at least during the EFind enhanced MapReduce job). Given the same index key, the lookup result will be the same.[2]

Under this assumption, we optimize the baseline strategy when there are many duplicate index lookup keys. Since index lookups often incur the overhead of remote communications, it is beneficial to reduce redundant index lookups. In the lookup cache strategy, we reduce redundant index lookups at a single machine node by implementing a lookup cache mechanism. EFind inserts the input ik and the result $\{iv\}$ of a $lookup$ operation into an LRU-organized cache. Before invoking the $lookup$ for another ik', it checks if ik' already exists in the cache. For a cache hit, EFind immediately returns the cached result. It invokes the $lookup$ method only when

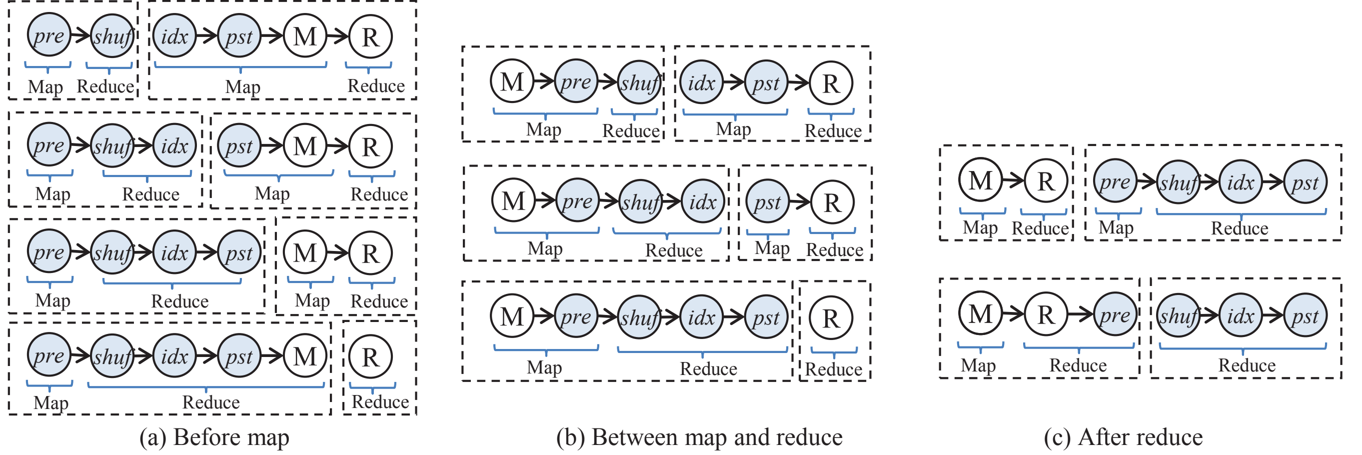
[1]In all the cost analyses, we assume a common case where the MapReduce computation and the indices are hosted in the same data center. The network bandwidth between two machines in the data center is BW. However, the analyses can be easily modified for other networking situations.

[2]Application developers can force EFind to use the baseline strategy if this assumption is false.



(a) Before map (b) Between map and reduce (c) After reduce

Figure 6: Baseline strategy for implementing EFind $IndexOperator$. (pre: $preProcess$, idx: $lookup$, pst: $postProcess$)



(a) Before map (b) Between map and reduce (c) After reduce

Figure 7: Re-partitioning strategy for implementing EFind $IndexOperator$. (pre: $preProcess$, shuf: shuffling, idx: $lookup$, pst: $postProcess$)

there is a miss in the lookup cache.

Let T_{cache} and R be the probe time and the miss ratio of the lookup cache, respectively. We can compute the cost of the lookup cache strategy as follows:

$$Cost_{cache} = N1 \cdot Nik_j [T_{cache} + R(\frac{Sik_j + Siv_j}{BW} + T_j)] \quad (2)$$

3.3 Re-partitioning Strategy

While the lookup cache strategy reduces redundancy at a machine node that performs index lookups, there can also be duplicate index keys *across multiple machine nodes*. As shown in Figure 7, we propose a re-partitioning strategy in order to reduce inter-machine redundancy.

The basic idea is to add a shuffling step between $preProcess$ and $lookup$ in order to group the index lookup requests with the same lookup key ik together, thereby removing inter-machine redundancy. The implementation introduces an additional MapReduce job and utilizes the group-by between Map and Reduce in this new MapReduce job for the shuffling step. We call this additional MapReduce job *shuffling job*.

From left to right in Figure 7, we show the cases where the $IndexOperator$ is before Map, between Map and Reduce, and after Reduce, respectively. In each case, we consider multiple ways to implement the strategy varying the boundary between the two MapReduce jobs. The boundary is significant because the results of the first job will be stored to the distributed file system. We would like to consider different boundaries in order to minimize the result size of the first job.

The cost of the re-partitioning strategy consists of three parts: the cost of shuffling, the overhead of storing and retrieving the results of the first job from the distributed file system, and the cost of the actual index lookups:

$$Cost_{repart} = Cost_{shuffle} + Cost_{result} + Cost_{lookup} \quad (3)$$

Here, the cost of shuffling mainly comes from transferring the output of $preProcess$:

$$Cost_{shuffle} = \frac{N1 \cdot Spre}{BW}$$

For $Cost_{result}$, we place the job boundary to minimize the result size of the first job:

$$Cost_{result} = f \cdot N1 \cdot S_{min}$$

$$S_{min} = \begin{cases} \min\{Spre, Sidx, Spost, Smap\}, \text{before Map} \\ \min\{Spre, Sidx, Spost\}, \text{between Map \& Reduce} \\ \min\{S1, Spre\}, \text{after Reduce} \end{cases}$$

Finally, suppose there are Θ duplicates per distinct index lookup key. We can compute $Cost_{lookup}$ as follows:

$$Cost_{lookup} = \frac{N1 \cdot Nik_j}{\Theta}(\frac{Sik_j + Siv_j}{BW} + T_j)$$

3.4 Index Locality Strategy

A distributed index often employs hash or range-based partition schemes. In many cases, it is possible to obtain the partition scheme from the distributed index. For example, the root of a distributed Btree describes the range partition scheme of the second level nodes. The meta-data server of a master-worker style index can often tell the partition scheme of the workers. It is also feasible to figure out the hash value intervals for machine nodes in the consistent hashing scheme in a Cassandra Key-Value store.

The partition scheme of an index can be communicated to EFind by implementing a partition method and setting a flag in the class of $IndexAccessor$. Then EFind will apply the partition method in the shuffling job of the re-partitioning strategy. As a result, the shuffling results (e.g. the lookup keys) will be co-partitioned as the index. This provides the basis for the index locality strategy.[3]

[3] A tempting idea is to try co-locating a lookup key partition i with its associated index partition i by setting Reducer i to be the machine hosting index partition i, thereby making index lookups local operations. Unfortunately, it is a bad idea to restrict a reducer to select only a single machine in a dynamic cloud environment because the unavailability of the machine can slow down the entire MapReduce job. Therefore, we do not assume the co-location of lookup keys and index partitions in this paper.

Let's consider the MapReduce job after the shuffling job in the above re-partition strategy (in the first case in Figure 7(a) and the first case in Figure 7(b)). There are two execution strategies:

- *Data Locality:* This is used by the original MapReduce. The system tries to schedule a Map task to run on a machine that hosts the main input data (e.g., index lookup keys). Then $lookup$ pays the overhead of remote communications to access the index.
- *Index Locality:* We can modify the MapReduce task scheduling mechanism to run a Map task on a machine that hosts an index partition. In this way, $lookup$ becomes local operations, saving network communication overhead. However, we need to transfer the main data to the machine.

The index locality strategy will be preferred if the result size of an index lookup is larger than the size of the lookup key and additional data in $(k1, v1)$. We compute the cost of the index locality strategy as follows:

$$Cost_{idxloc} = Cost_{shuffle} + Cost_{result} + Cost'_{lookup} \quad (4)$$

where $Cost_{shuffle}$ and $Cost_{result}$ are the same as computed in Section 3.3. $Cost'_{lookup}$ is computed as follows:

$$Cost'_{lookup} = \frac{N1 \cdot Nik_j}{\Theta} T_j + \frac{N1 \cdot Spre}{BW}$$

3.5 Multiple Indices in a Single $IndexOperator$

Having discussed execution strategies for a single index, we now consider multiple independent indices in an $IndexOperator$. From the cost analysis of the four index access strategies, we observe the following four properties:

- *Property 1:* The costs of the baseline strategy and the lookup cache strategy of an index j are not affected by the order to access the multiple indices.
- *Property 2:* The cost of the re-partitioning strategy and the index locality strategy depend on the order to access multiple indices because the data to be shuffled must contain all the results of earlier index lookups.
- *Property 3:* If the order to access multiple indices is fixed, then the costs of index access strategies for an index j are not affected by the strategy choices of the other indices.
- *Property 4:* In an optimal plan, indices with the re-partitioning strategy or the index locality strategy (if any) will be accessed before those with the baseline or the lookup cache strategy.

Property 1 and 2 are clear from the cost formulae. For Property 3, we note that the result size of an index lookup will be the same regardless of the choice of index access strategies. Therefore, if the order of accessing multiple indices is fixed, all the size parameters in the cost formulae will be fixed, and therefore the costs for a given index will be fixed.

Property 4 results from the fact that the intermediate data size always increases as EFind includes index lookup results $\{iv_j\}$ into the output $(k1', v1', \{\{ik_1\}, ..., \{ik_m\}\})$ of $preProcess$. The larger the intermediate data size, the larger the costs for the re-partitioning strategy and the index locality strategy, while the costs of the other two strategies are not affected. Therefore, to minimize the total cost, the optimal plan will access indices with the re-partitioning strategy or the index locality strategy first.

Given these four properties, we propose the following two algorithms for computing the index access strategies for multiple independent indices in an $IndexOperator$:

- *Algorithm FullEnumerate:* Given m independent indices, the algorithm generates all $m!$ orders to access the indices. For a given order, it determines the index access strategy for each index using Property 3. Once the strategy of an index is determined to be either baseline or lookup cache, the algorithm would only consider baseline/lookup cache strategies for the rest of indices (Property 4). After computing the cost for each index access order, the algorithm chooses the index access order with the minimum cost.
- *Algorithm k-Repart:* Given m independent indices, the algorithm generates all the k-permutations of the m independent indices. For each generated k-permutation, it computes the strategies for the first k indices as in Algorithm FullEnumerate, while only considering the baseline or the lookup cache strategies for the rest $(m - k)$ indices. After computing the cost for each index access order, the algorithm chooses the index access order with the minimum cost.

Algorithm FullEnumerate computes $m!$ plans, while Algorithm k-Repart computes $P(m, k) = m(m-1)\cdots(m-k+1)$ plans. In a typical situation, the number of indices at an $IndexOperator$ is small. For example, $m \leq 5$, $m! \leq 120$. It is feasible to employ Algorithm FullEnumerate to compute the optimal plan. When m is very large, Algorithm FullEnumerate may be too expensive to use. We fall back to Algorithm k-Repart with a small k, e.g., 1-Repart or 2-Repart. Note that the cost of adding an extra MapReduce job by the re-partitioning or the index locality strategies can be high, thus it is rare that such strategies are chosen by many indices. Therefore, Algorithm k-Repart with a small k often generates a good plan.

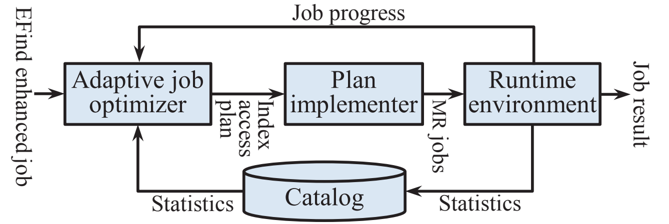


Figure 8: EFind runtime system overview.

4. ADAPTIVE OPTIMIZATION FOR INDEX ACCESS

Having discussed how to optimize index access if sufficient and accurate statistics about the indices are available, we switch our attention to applying these techniques in optimizing EFind enhanced jobs in this section.

One approach is to record statistics at the end of a job, and then use the statistics collected from previous jobs to statically compute an index access plan for a new job. However, in practice, a statically generated plan can be sub-optimal for two reasons: (i) statistics (for a new index or a new $IndexOperator$) may be missing; and (ii) statistics collected from previous MapReduce jobs may not be accurate for the current job if the data use patterns differ. Compared to traditional databases, the latter concern is more significant in the MapReduce system because the extensive use of user-defined functions in MapReduce limits the system's power to understand the data use patterns.

Given these considerations, we design an EFind runtime system to support adaptive optimization for index access. In the following, we first overview the system in Section 4.1. Then we discuss how to collect statistics and how to change the index access plan for an ongoing job in details in Section 4.2 and 4.3, respectively.

4.1 Overview of Adaptive Optimization

Figure 8 depicts the components of the EFind runtime system. First, the *adaptive job optimizer* generates an initial index access

Algorithm 1 Dynamic Re-Optimization

```
Input: Job currJob, Plan currPlan
 1: for each Collected Statistics do
 2:    if Standard deviation / Mean > Threshold then
 3:       return null;
 4: IndexOperatorSet idx_ops ← null;
 5: if currJob.isAtMapPhase() then
 6:    idx_ops ← getIdxOpsBeforeReduce(currJob);
 7: else
 8:    idx_ops ← getIdxOpsAfterReduce(currJob);
 9: newPlan ← optimize(idx_ops);
10: if currPlan.cost - newPlan.cost < planChangeCost then
11:    newPlan ← null;
12: return newPlan;
```

plan, and re-optimizes the plan if necessary during runtime. Using the cost formulae and the algorithms as described in Section 3, it finds the best index access strategies based on the statistics as recorded in the catalog. Second, the *plan implementer* implements the chosen index access plan by inserting chained functions into the MapReduce job and/or by adding shuffling MapReduce jobs (e.g., for the re-partitioning strategy). Third, the *runtime environment* enhances the original MapReduce runtime by collecting index access statistics. In addition, when a new plan is generated for an on-going job J, the plan implementer and the runtime environment collaborate to stop Job J at an appropriate point, submit a new job according to the new index access plan, and try to reuse the intermediate results of Job J as much as possible in the new job. Finally, the *catalog* stores statistics about index accesses.

Since the characteristics of one Map (Reduce) task is often representative of that of all the other Map (Reduce) tasks, EFind updates the statistics in the catalog after a Map task or a reduce task completes. Note that when processing a large amount of data, it is typical to configure a MapReduce job to use much larger number of Map tasks than the number of machine nodes so that Map tasks are performed in multiple rounds. The original intention is to start transferring the Map outputs to Reducers after the first round of Map tasks, thereby overlapping the data transferring time with the Map computation. We take advantage of such job structure in that the statistics collected from the tasks in the first round of Map may trigger re-optimization of the job and the rest of the computation will be better optimized.

Algorithm 1 shows the algorithm for dynamically re-optimize a running job. The algorithm performs re-optimization only when the variance of the collected statistics is small enough (line 1–3). Otherwise, the statistics may not represent the characteristics of the entire job. (We will discuss this point in more details in Section 4.2.) If the current running job ($currJob$) is at the Map phase (line 5), the algorithm extracts $IndexOperator$(s) (*idx_ops*) that appear before the Reduce phase (line 6). EFind will ignore the $IndexOperator$(s) at the Reduce phase because their statistics have not yet been updated. Otherwise, the current running job is at the Reduce phase, and the algorithm extracts the $IndexOperator$(s) that appear after the Reduce tasks (line 8). EFind ignores the operators at the Map phase because the Map phase has already completed. Then, the algorithm invokes the adaptive runtime optimizer to re-optimize *idx_ops* based on the up-to-date statistics (line 9). If the generated new plan significantly improves the current plan such that the improvement is larger than the overhead to perform the plan change, the algorithm returns the new plan. Otherwise, it returns null and EFind continues with the current plan.

We will change the execution plan of a job at most *once* in order to keep the overhead of plan changes low. Note that we expect the runtime optimizer (line 9) to find the best plan when statistics are representative of the entire job.

4.2 Collecting Statistics during Execution

EFind collects statistics on the fly and updates the catalog whenever a Map task or a Reduce task completes. We leverage a feature in MapReduce systems, called *counter*, in the implementation. A counter can be incremented by individual Map or Reduce tasks and will be globally visible. We create a set of counters at the following places as the basis to obtain the statistics in Table 1:

- *preProcess:* We add counters to record the number of inputs, the total input size, the number of keys extracted for each index, and the total output size. Based on these counters, we can calculate $N1$, $S1$, Nik_j, and $Spre$, respectively.
- *lookup:* We add counters to record the total input size and the total output size. Based on these counters, we can calculate Sik_j and Siv_j, respectively.
- *postProcess:* We add a counter to record the total output size in order to compute $Spost$.
- *Map:* We add a counter to record the total output size in order to compute $Smap$.

Moreover, statistics such as network bandwidth (BW), average time for storing and retrieving from the distributed file system (f), and average time for a probe in the lookup cache (T_{cache}) are straightforward to measure offline.

Furthermore, we consider how to obtain average time for index j to serve a lookup (T_j) and miss ratio of lookup cache (R). For T_j, we sample the time to perform an index lookup, and subtract the estimated network communication cost ($\frac{Sik_j+Siv_j}{BW}$) from the time. The result is accumulated and averaged to obtain T_j. For estimating R, we use a simple version of the lookup cache that does not cache lookup results, and sample significantly long (e.g., 100x of the cache size) sequences of lookups.[4]

Finally, we consider the most challenging statistics to obtain in Table 1 — Θ, i.e. average number of duplicates per index lookup key. We apply the Flajolet and Martin's (FM) algorithm [9], which estimates the number of distinct values in data streams. For each Map or Reduce task, we keep a FM bit vector updated by the lookup keys. Local FM bit vectors are OR-ed together to compute the distinct number of keys across machines. Then we divide the total number of lookup keys by the estimated number of globally distinct keys to obtain Θ.

Variance of Collected Statistics. Re-optimization makes sense only when the collected statistics reflect the characteristics of the job. We consider the statistics collected at each Map or Reduce task as a random sample. Note that according to the central limit theorem, we can assume the normal distribution when the number of samples is large. Then with a probability of 99%, the sample mean is within 3 times the standard deviation from the true mean. Therefore, a small variance across the samples gives a high confidence that the computed sample mean is close to the true mean.

For each type (y) of statistics on *preProcess*, *lookup*, and *postProcess*, we compute the sample variance (S^2) of the statistics across different Map or Reduce tasks.

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y}) = \frac{1}{n-1}\sum_{i=1}^{n} y_i^2 - \frac{n}{n-1}\bar{y}^2 \qquad (5)$$

where y_i is the local statistics computed at task i, n is the number of Map or Reduce tasks from which we collect statistics, and $\sum y_i^2$

[4]Note that the lookup cache size is fixed in our implementation. We leave the study of varying lookup cache sizes to future work.

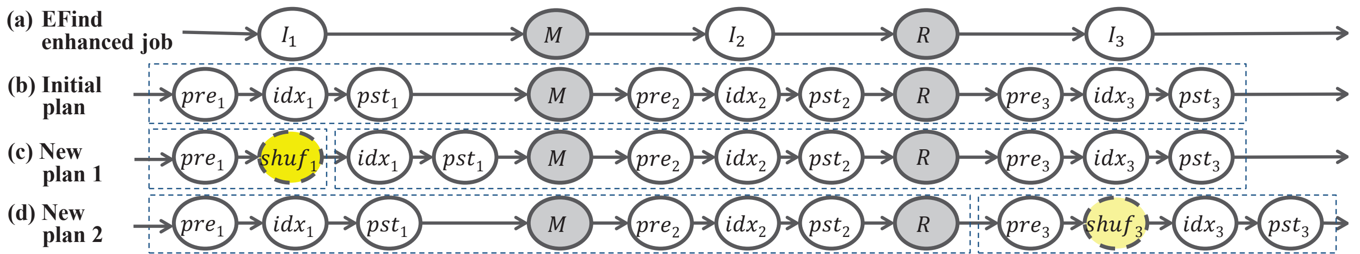
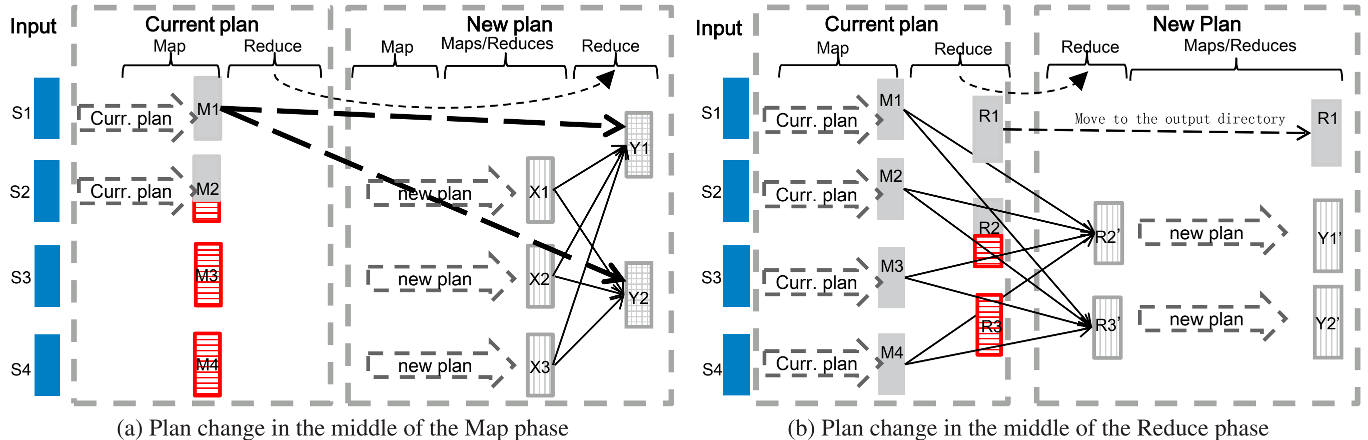


Figure 9: Adaptive optimization for an EFind enhanced job. (Note: a dotted rectangle shows a single MapReduce job.)



(a) Plan change in the middle of the Map phase

(b) Plan change in the middle of the Reduce phase

Figure 10: Dynamically changing index access plans while reusing intermediate results.

is accumulated using a counter. The sample standard deviation is the square root of S^2. In Algorithm 1, we make sure that the standard deviation over mean is below a threshold (e.g., 0.05) before performing re-optimization.

4.3 Changing the Index Access Plan of an Ongoing Job

A naïve approach kills the current MapReduce job, discards all the intermediate results, then restarts the job with the new plan. However, it wastes all the previous computation.

We would like to reuse previously computed results as much as possible. Figure 9 and Figure 10 illustrate our design for dynamically changing index access plans. Figure 9(a) shows an EFind enhanced job with three $IndexOperator$s I_1, I_2, and I_3. Figure 9(b) depicts the initial plan that employs the baseline strategy. Figure 9(c) and (d) show two cases, changing index access plans either in the middle of the Map phase or in the middle of the Reduce phase. As discussed previously in Figure 7, the two new plans employ the re-partitioning strategy for I_1 and I_3, respectively.

When EFind decides to change the plan in the middle of the Map or Reduce phase, a task can be in one of the following three states: (1) it has already completed (e.g., M_1 in Figure 9(c), R_1 in Figure 9(d)); (2) it is being processed with partial results (e.g., M_2 in Figure 9(c), R_2 in Figure 9(d)); or (3) it has not yet started (e.g., M_3 and M_4 in Figure 9(c), R_3 in Figure 9(d)). It is almost free to reuse results from completed tasks. For a completed Map task M, EFind only needs to configure all the Reduce tasks to obtain intermediate results from M (in addition to the new Map tasks in the new plan). For a completed Reduce task R, EFind only needs to specify that the output of R should be part of the job output. Note that in both cases, there is no need to transfer any significant amount of data or perform significant computation. On the other hand, it can be very expensive to merge the results of a partially completed task with those from the new plan in order to guarantee that each result is produced once and exactly once. Based on these considerations, EFind reuses results from completed tasks in state (1), but applies the new plan to tasks in state (2) or state (3).

We describe how EFind changes the index access plan:

- *Plan Change in the Middle of the Map Phase:* As shown in Figure 10(a), EFind performs two steps for the plan change. First, it applies the new plan to the input splits that have not completed (e.g., S_2, S_3, and S_4). Second, it runs the Reduce tasks of the new plan (e.g., Y_1 and Y_2). The Reduce tasks retrieve outputs not only from the new Map tasks (e.g., X_1, X_2, and X_3), but also from the completed tasks in the old plan (e.g., M_1).
- *Plan Change in the Middle of the Reduce Phase:* As shown in Figure 10(b), to apply the new plan, EFind runs the new Reduce tasks (e.g., R'_2 and R'_3), merges the results of the new tasks (e.g., R'_2 and R'_3) with those of previously completed tasks (e.g., R_1), and perform any subsequent processing.

5. EXPERIMENTS

In this section, we empirically evaluate EFind using both real-world and synthetic data sets.

5.1 Experimental Setup

Cluster setup. We perform the experiments on a cluster of 12 nodes. Every node is an HP ProLiant BL460c blade server equipped with two Intel(R) Xeon(R) X5675 64-bit 3.07GHz CPUs (6 cores/12 threads, 12MB cache), 96GB memory, and a 1TB 7200rpm HP SAS hard drive, running 64bit Red Hat Enterprise Linux Server release 6.2 with Linux 2.6.32-220 kernel. The blade servers are connected through a 1Gbps Ethernet switch.

We implemented EFind on top of Apache Hadoop 1.0.0. In our Hadoop cluster, 1GB memory is allocated for each Hadoop daemon. One TaskTracker and one DataNode daemon run at each worker node. Every TaskTracker is configured to run up to 8 Map

and 4 Reduce tasks by default. The DFS chunk size and replication factor are 64MB and 3, respectively. The lookup cache contains up to 1024 index key-value entries. We use Oracle Java 1.7 in our experiments.

Index. Our experiments use Apache Cassandra 1.0.6 [4] to provide index services unless otherwise noted. Our 12-node Cassandra setup is running with Hadoop in the same cluster. To control the partition of data and be aware of the partition locations, we use the NetworkTopologyStrategy to control the partition placement among nodes. PropertyFileSnitch is used as the endpoint_snitch, so that we can explicitly control the network topology and data distribution. The index is divided into 32 partitions using the HashPartitioner of Apache Hadoop. One index partition is replicated to three data nodes. The partition information is stored in a column family, which is replicated to every data node.

Data sets and Data Analysis Jobs. We evaluate the performance of EFind using the following four data sets:

- ***LOG***: LOG contains a set of real-world web log traces from a popular web site. An event record consists of: event ID, timestamp, source IP, visited URL, and up to 7 other fields. The data set contains 15 million events and is 7GB large. The application computes the top-k frequently visited URLs in each geographical region. It uses a cloud service to look up the geographical region for an IP address. The cloud service runs on a single node with Java RMI interface.
- ***TPC-H***: We generate a TPC-H [20] data set with a scale factor of 10, and perform Q3 and Q9 in the experiments. We compose MapReduce jobs to follow the same join order as MySQL. For Q3, the job first joins LineItem with Orders, then with Customer. For Q9, the job first joins LineItem with Supplier, then with Part, PartSupply, Orders, and finally with Nation. The main input for both Q3 and Q9 is the LineItem table. We create indices on the rest of the tables in the queries, and essentially perform index nested-loop joins. Moreover, we also perform a set of experiments while duplicating the LineItem table 10 times, which are represented as TPC-H DUP10.
- ***Synthetic***: The synthetic data set contains 10 million records. Each record consists of an integer key and a 1KB-sized value. The keys are uniformly randomly generated from [0, 4999999]. We build an index that maps each distinct key to an index value of size l, and run a job to join the data set with the index. We vary the parameter l in the experiments to model different workload characteristics.
- ***OSM***: We compare an EFind based k-nearest neighbor join (kNNJ) implementation with a hand-tuned implementation [22] using a real-world OpenStreetMap (OSM) data set. The data set contains about 42 million records of geographic locations in the US, and is 14.8GB large. Every record consists of a record ID and a 2D coordinate. The job computes kNNJ ($k = 10$) between two randomly selected sub-sets (A and B) of records, each with 40 million records, from the OSM data set. For the EFind based implementation, we use A as the main input to MapReduce and build a distributed index on B to support kNN search. We partition the US map into 4×8 cells with small overlapping regions, then build an R* tree for each cell. Each R* tree is replicated to 3 machines.

Index Access Strategies. For each job, we run six experiments: (1) the baseline strategy (*Base*), (2) the lookup cache strategy (*Cache*), (3) the re-partitioning strategy (*Repart*), (4) the index locality strategy (*Idxloc*), (5) dynamic adaptive optimization (*Dynamic*), and (6) static optimization with sufficient statistics (*Optimized*). For (5), we start the job with no statistics, and perform re-optimization on the fly. Usually, in the first round of Map tasks, EFind collects sufficient statistics to perform re-optimization. Then EFind may perform dynamic plan change if a better new plan is generated. We measure and report the execution time of the jobs.

5.2 Index Access Strategies

We first evaluate each index access strategy using LOG, TPC-H, and Synthetic data sets.

LOG. We would like to understand the performance of LOG under various delay conditions for looking up an IP address. The cloud service incurs a $t = 0.8ms$ delay for a lookup. As shown on the X-axis in Figure 11(a), we introduce an extra $0, 1ms, \cdots, 5ms$ to the lookup. From Figure 11(a), we see that (i) compared to baseline, the lookup cache strategy achieves a factor of 1.2–1.5x improvements; (ii) the re-partitioning strategy achieves an additional factor of 1.4–2.2x speedups over the lookup cache strategy; and (iii) the improvements are larger with longer delays.

From the log traces, we find that an IP often visits multiple URLs in a short period of time. The visits are often served by two or more web servers, and recorded in two or more log files. Different log files are processed in different Map tasks. Consequently, we see both strong local redundancy and strong cross-machine redundancy in index lookups, which explain the effectiveness of the lookup cache and the re-partitioning strategies. Moreover, the benefits become larger with longer lookup delays because the savings of reducing redundant lookups increase.

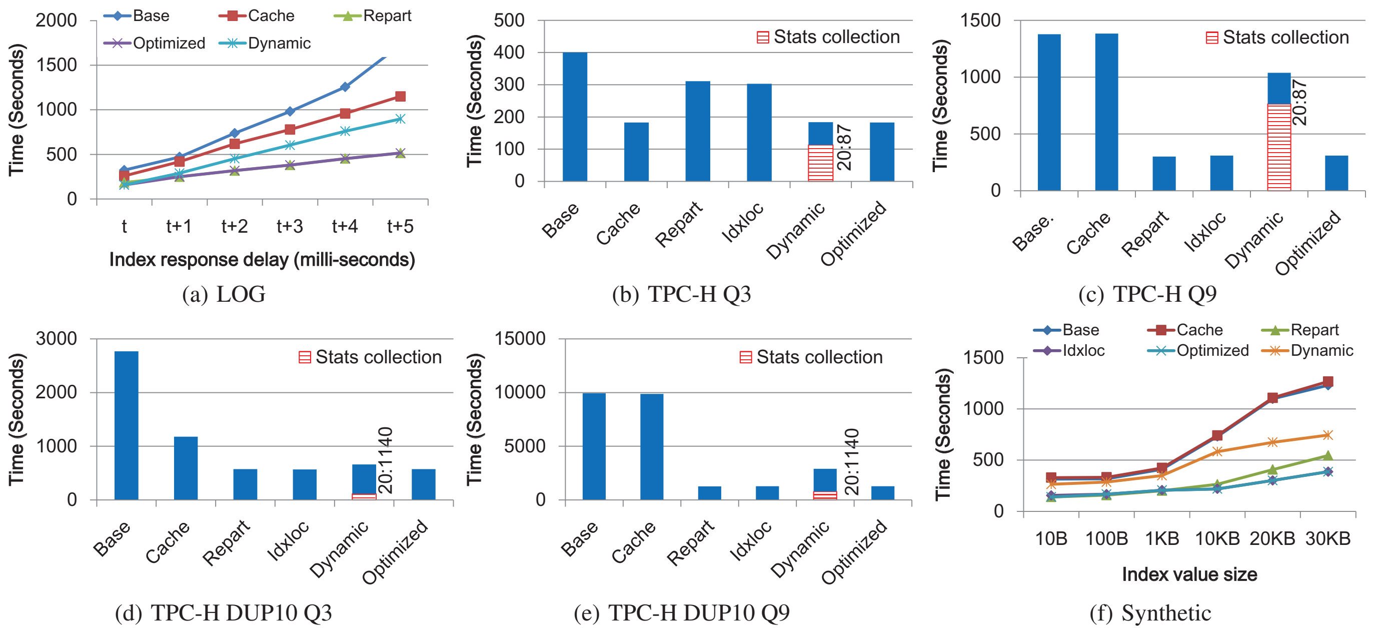
Note that index locality does not apply to LOG because the cloud service is located on a single machine.

TPC-H. Figure 11(b)-(e) report the experimental results for TPC-H Q3 and Q9. First, we see that the lookup cache strategy achieves 2.2–2.4x improvements over baseline for Q3, but has little benefits for Q9. In Q3, LineItem records are used to look up the index on Orders. Since the LineItem records that is associated with the same order record are stored consecutively in the TPC-H data set, there is high local redundancy in index lookups, which is effectively optimized by the lookup cache strategy. On the other hand, Q9 first probes the index on Supplier using LineItem records. There is no locality in the index lookup. Hence the cache sees very high miss rate.

Second, for re-partitioning, we choose one of the indices with the most benefits to apply re-partitioning (i.e., Orders in Q3, Supplier in Q9), while using the lookup cache strategy for the rest. As shown in Figure 11(c), re-partitioning reduces the running time of baseline by a factor of 4.6x. This is because LineItem records with the same supplier are grouped together after re-partitioning, effectively removing all redundant index accesses for the index on Supplier. As shown in Figure 11(e), TPC-H DUP10 duplicates the LineItem table 10 times. This leads to 10x redundant index keys, and thus higher benefits from re-partitioning, a 7.9x speedup over baseline.

Third, as shown in Figure 11(b), re-partitioning has worse performance than the lookup cache strategy for Q3. As explained in the above, the lookup cache already effectively removes most of the redundancy. Therefore, it is not worthwhile to pay the cost of re-partitioning. As shown in Figure 11(d), TPC-H DUP10 introduces 10x redundancy across machines, and re-partitioning removes this global redundancy, achieving 2.1x improvements over the lookup cache strategy for Q3.

Finally, for TPC-H Q3 and Q9, index locality does not show clear benefits over re-partitioning. In fact, it is even slightly worse than re-partitioning in some cases. The reason is that the number of index lookups has already been significantly reduced with re-



(a) LOG (b) TPC-H Q3 (c) TPC-H Q9

(d) TPC-H DUP10 Q3 (e) TPC-H DUP10 Q9 (f) Synthetic

Figure 11: Experimental results for LOG(a), TPC-H ($b - c$), TPC-H DUP10 ($d - e$), and Synthetic (f).

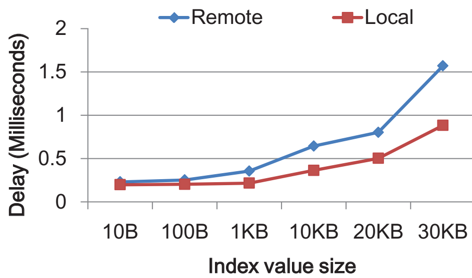


Figure 12: Index lookup latency for Synthetic.

partitioning. The cost of remote index lookup is actually lower than the cost of transferring input splits to co-locate with the index partitions.

Synthetic. We vary the size of index lookup result and study its impact on index access performance. Figure 12 shows the elapsed time of a local vs. remote index lookup while varying the size of lookup result from 10B to 30KB. We see that as the result size increases, both the index access time and the gap between local and remote index accesses increase. The latter is because more data need to be transferred over network.

Figure 11(f) compares the index access strategies while varying index lookup result size. First, the lookup cache strategy sees little benefits. This is because the lookup keys are randomly generated from 5 million distinct values, and incur very high miss rate in the 1024-entry lookup cache.

Second, there are 10 million keys in total, and thus on average every key occurs twice in the data set. Re-partitioning can effectively group the redundant keys together, reducing redundant index lookups. It achieves 2.0x–2.8x speedups compared to baseline.

Finally, index locality can achieve even better performance than re-partitioning. It plays a more significant role when the result size is large, and the remote index lookup cost is high. When the result size is 1KB and smaller, the input transfer overhead dominates, and index locality is slightly worse than re-partitioning. When the result size is larger than 1KB, it is more beneficial to remove remote index lookups, and index locality achieves 1.2–1.4x improvements over re-partitioning.

5.3 Adaptive Optimization

In LOG, TPC-H, and Synthetics, the optimal performance is achieved by different index access strategies (i.e. cache, re-partitioning, or index locality). In the following, we consider two cases: (i) optimization with sufficient statistics (*Optimized*); and (ii) adaptive optimization starting with no statistics (*Dynamic*). We compare optimized and dynamic with the optimal performance.

LOG and Synthetic. First, in Log (Figure 11(a)) and Synthetic (Figure 11(f)), we see that the running time of optimized is the same as the optimal performance. EFind correctly selects the optimal plan with sufficient statistics. Second, dynamic starts running with the baseline strategy while collecting statistics. Then EFind dynamically re-optimizes the jobs. We find that EFind chooses the correct optimal plans after re-optimization. Due to the overhead of the statistics collection phase, dynamic is slower than the optimal performance, but it is significantly faster than both baseline and cache.

TPC-H. First, we see that the running time of optimized is the same or very close to the optimal performance. When the costs of re-partitioning and index locality are very close, EFind may choose a plan different from the optimal plan. Note that cost estimation based on statistics and simple formulae may not be fully accurate.

Second, the adaptive optimization, dynamic, always achieves better performance than baseline in all TPC-H experiments. However, because of the statistics collection phase, dynamic is slower than the optimal performance. Specifically, for Q9 in Figure 11(c), the statistics collection phase is the first round of 20 Map tasks, which takes 765 seconds. After re-optimization, the rest of the computation with 87 Map tasks and the reduce phase takes only 273 seconds. This effect will be reduced when many Map tasks are used to process a large amount of data. As shown in Figure 11(d) and Figure 11(e), if we increase the LineItem table by 10 times, the statistics collection phase consists of only a small portion of the total runtime. Therefore, the performance of dynamic is very close to the optimal performance.

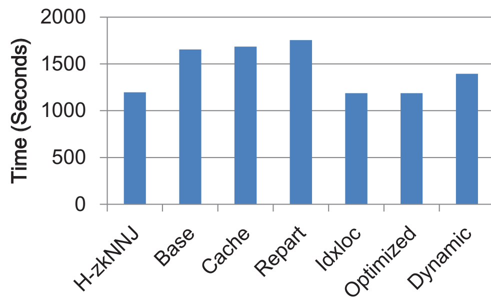


Figure 13: Comparing EFind solutions (Base, Cache, Repart, Idxloc, Optimized, Dynamic) with a hand-tuned implementation (H-zkNNJ) for k-nearest neighbor join (kNNJ).

5.4 Comparing EFind with Hand-Tuned Implementation

Finally, we use k-nearest neighbor join as an example to compare an EFind-based solution with a hand-tuned implementation on MapReduce. The hand-tuned implementation of kNNJ is H-zkNNJ [22]; We set its two parameters $\alpha = 2$ and $\epsilon = 0.003$. Our EFind implementation performs an index nested-loop join between the two sets of locations. As shown in Figure 13, EFind-based solution (with index locality as the optimal strategy) achieves similar performance as the hand-tune implementation. Using EFind, application developers can easily and flexibly express operations involving indices, while achieving very good performance.

6. CONCLUSION

A wide range of big data processing applications require selectively accessing one or many data sources other than the main input of MapReduce. We model such a data source as an index. In this paper, we present an EFind index access interface to easily and flexibly incorporate index access into MapReduce. We propose and analyze a set of index access strategies, and design a runtime system to support adaptive optimization. From experimental results on four real-world and synthetic data sets, we see that (i) the lookup cache, re-partitioning, and index locality strategies can each be the optimal strategy under certain application scenarios; (ii) when statistics are sufficient, EFind chooses the optimal plan or a plan that has very similar performance to the optimal, achieving 2–8x speedups over the baseline strategy; (iii) when there are no statistics, EFind starts a job with the baseline strategy, collects statistics on the fly, and re-optimizes the job to choose a better plan; and (iv) an EFind-based solution can achieve similar performance as a hand-tuned implementation. In conclusion, EFind provides a flexible programming interface and a capable runtime system to automatically achieve good performance for accessing indices in MapReduce.

7. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.

[2] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1(1):598–609, Aug. 2008.

[3] Apache hadoop project. `http://hadoop.apache.org/`.

[4] Apache cassandra project. `http://http://cassandra.apache.org/`.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, June 2008.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] 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). *Proc. VLDB Endow.*, 3(1-2):515–529, Sept. 2010.

[8] S. Feng, Y. Xiong, C. Yao, L. Zheng, and W. Liu. Acronym extraction and disambiguation in large-scale organizational web pages. In *CIKM*, pages 1693–1696, 2009.

[9] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, Sept. 1985.

[10] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[11] IDC. The digital universe in 2020: Big data, bigger digital shadows, biggest growth in the far east. http://www.emc.com/leadership/digital-universe/, 2012.

[12] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of mapreduce: an in-depth study. *Proc. VLDB Endow.*, 3(1-2):472–483, Sept. 2010.

[13] P. Jiang, H. Hou, L. Chen, S. Chen, C. Yao, C. Li, and M. Wang. Wiki3c: exploiting wikipedia for context-aware concept categorization. In *WSDM*, pages 345–354, 2013.

[14] J. Lin, G. Xiang, J. I. Hong, and N. M. Sadeh. Modeling people's place naming preferences in location sharing. In *UbiComp*, pages 75–84, 2010.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[16] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, 1989.

[17] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD*, pages 165–178, 2009.

[18] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[19] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[20] Tpch focus. `http://http://www.tpc.org/tpch/`.

[21] M. Ye, D. Shou, W.-C. Lee, P. Yin, and K. Janowicz. On the semantic annotation of places in location-based social networks. In *KDD*, pages 520–528, 2011.

[22] C. Zhang, F. Li, and J. Jestes. Efficient parallel knn joins for large data in mapreduce. In *EDBT*, pages 38–49, 2012.

[23] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, Dec. 1998.