Opportunistic Physical Design for Big Data Analytics

Jeff LeFevre† Jagan Sankaranarayanan∗ Junichi Tatemura∗ Neoklis Polyzotis† Michael J. Carey‡
∗NEC Labs America, Cupertino, CA †University of California, Santa Cruz ‡University of California, Irvine
{jlefevre,alkis}@cs.ucsc.edu, {jagan,hakan,tatemura}@nec-labs.com, mjcarey@ics.uci.edu

ABSTRACT

Big data analytical systems, such as MapReduce, perform aggressive materialization of intermediate job results in order to support fault tolerance. When jobs correspond to exploratory queries submitted by data analysts, these materializations yield a large set of materialized views that we propose to treat as an opportunistic physical design. We present a semantic model for UDFs that enables effective reuse of views containing UDFs along with a rewrite algorithm that provably finds the minimum-cost rewrite under certain assumptions. An experimental study on real-world datasets using our prototype based on Hive shows that our approach can result in dramatic performance improvements.

1. INTRODUCTION

Data analysts have the crucial task of analyzing the ever increasing volume of data that modern organizations collect in order to produce actionable insights. As expected, this type of analysis on big data is highly exploratory in nature and involves an iterative process: the data analyst starts with an initial query over the data, examines the results, then reformulates the query and may even bring in additional data sources, and so on [9]. Typically, these queries involve sophisticated, domain-specific operations that are linked to the type of data and the purpose of the analysis, e.g., performing sentiment analysis over tweets or computing network influence. Because a query is often revised multiple times in this scenario, there can be significant overlap between queries. There is an opportunity to speed up these explorations by reusing previous query results either from the same analyst or from different analysts performing a related task.

MapReduce (MR) has become a de-facto tool for this type of analysis. It offers scalability to large datasets, easy incorporation of new data sources, the ability to query right away without defining a schema up front, and extensibility through user-defined functions (UDFs). Analyst queries are often written in a declarative query language, e.g., HiveQL or PigLatin, which are automatically translated to a set of MR jobs. Each MR job involves the materialization of intermediate results (the output of mappers, the input of reducers and the output of reducers) for the purpose of failure recovery. A typical Hive or Pig query will spawn a multi-stage job that will involve several such materializations. We refer to these execution artifacts as opportunistic materialized views.

We propose to treat these views as an opportunistic physical design and to use them to rewrite queries. The opportunistic nature of our technique has several nice properties: the materialized views are generated as a by-product of query execution, i.e., without additional overhead; the set of views is naturally tailored to the current workload; and, given that large-scale analysis systems typically execute a large number of queries, it follows that there will be an equally large number of materialized views and hence a good chance of finding a good rewrite for a new query. Our results indicate the savings in query execution time can be dramatic: a rewrite can reduce execution time by up to an order of magnitude.

Rewriting a query using views in the context of MR involves a unique combination of technical challenges that distinguish it from the traditional problem of query rewriting. First, the queries and views almost certainly contain UDFs, thus query rewriting requires some semantic understanding of UDFs. These MR UDFs for big data analysis are composed of arbitrary user-code and may involve a sequence of MR jobs. Second, any query rewriting algorithm that can utilize UDFs now has to contend with a potentially large number of operators since any UDF can be included in the rewriting process. Third, there can be a large search space of views to consider for rewriting due to the large number of materialized views in the opportunistic physical design, since they are almost free to retain (storage permitting).

Recent methods to reuse MR computations such as ReStore [6] and MRShare [21] lack any semantic understanding of execution artifacts and can only reuse/share cached results when execution plans are syntactically identical. We strongly believe that any truly effective solution will have to incorporate a deeper semantic understanding of cached results and “look into” the UDFs as well.

Contributions. In this paper we present a novel query-rewrite algorithm that targets the scenario of opportunistic materialized views in an MR system with queries that contain UDFs. We propose a UDF model that has a limited semantic understanding of UDFs, yet enables effective reuse of previous results. Our rewrite algorithm employs techniques inspired by spatial databases (specifically, nearest-neighbor searches in metric spaces [12]) in order to provide a cost-based incremental enumeration of the huge space of candidate rewrites, generating the optimal rewrite in an efficient manner. Specifically, our contributions can be summarized as follows:

- A gray-box UDF model that is simple but expressive enough to capture a large class of MR UDFs that includes many common analysis tasks. The UDF model further provides a quick way to compute a lower-bound on the cost of a potential rewrite given just the query and view definitions. We provide the model and the types of UDFs it admits in Sections 3–4.

- A rewriting algorithm that uses the lower-bound to (a) gradually explode the space of rewrites as needed, and (b) only attempts a rewrite for those views with good potential to produce a low-cost
2. PRELIMINARIES

Here we present the architecture of our system. We first briefly describe its components and how they interact with one another. We then provide the notations and problem definition.

2.1 System Architecture

Figure 1 provides a high level overview of our system and its components. Our system is built on top of Hive, and queries are written in HiveQL. Queries are posed directly over log data stored in HDFS. In Hive, MapReduce UDFs are given by the user as a series of Map or Reduce tasks containing arbitrary user code expressed in a supported language such as Java, Perl, Python, etc. To reduce execution cost, our system automatically rewrites queries based on the existing views. To facilitate this, our system collects statistics by running a lightweight Map job that materializes views currently in the system such as the view definition, the materialized views. These views are stored in the system (space permitting) as the opportunistic physical design.

During query execution, all by-products of query processing (i.e., the intermediate materializations) are retained as opportunistic materialized views. These views are stored in the system (space permitting) as the opportunistic physical design.

The materialized view metadata store contains information about the materialized views currently in the system such as the view definitions and standard data statistics used in query optimization. For each view stored, we collect statistics by running a lightweight Map job that samples the view’s data. This constitutes a small overhead, but as we show experimentally in Section 8, this time is a small fraction of query execution time.

The rewriter, presented in Section 6, uses the materialize view metadata store to rewrite queries based on the existing views. To facilitate this, our optimizer generates plans with two types of annotations on each plan node: (1) the logical expression of its computation (Section 3.2) and (2) the estimated execution cost (Section 4.2).

The rewriter uses the logical expression in the annotation when searching for rewrites for each node in the plan. The expression consists of relational operators or UDFs. For each rewrite found during the search, the rewriter utilizes the optimizer to obtain an estimated cost for the rewritten plan.

2.2 Notations

$W$ denotes a plan generated by the query optimizer, which is represented as a DAG containing $n$ nodes, ordered topologically. Each node represents an MR job. We denote the $i^{th}$ node of $W$ as $NODE_i$, $i \in [1, n]$. The plan has a single sink that computes the result of the query; under the topological order assumption the sink is $NODE_n$. $W_i$ is a sub-graph of $W$ containing $NODE_i$ and all of its ancestor nodes. We refer to $W_i$ as one of the rewritable targets of plan $W$. As is standard in Hive, the output of each job is materialized to disk. Hence, a property of $W_i$ is that it represents a materialization point in $W$. In this way, materializations are free except for statistics collection. An outgoing edge from $NODE_i$ to $NODE_j$ represents data flow from $i$ to $j$. $V$ is the set of all opportunistic materialized views in the system.

We use $\text{Cost}(NODE_i)$ to denote the cost of executing the MR job at $NODE_i$, as estimated by the query optimizer. Similarly, $\text{Cost}(W_i)$ denotes the estimated cost of running the sub-plan rooted at $W_i$, which is computed as $\text{Cost}(W_i) = \sum_{\forall NODE_k \in W_i} \text{Cost}(NODE_k)$.

We use $r_i$ to denote an equivalent rewrite of target $W_i$, i.e., $r_i$ uses only views in $V$ as input and produces an identical output to $W_i$, for the same database instance $D$. A rewrite $r^*$ represents the minimum cost rewrite of $W$ (i.e., target $W_n$).

2.3 Problem Definition

Given these basic definitions, we introduce the problem we solve in this paper.

**Problem Statement.** Given a plan $W$ for an input query $q$, and a set of materialized views $V$, find the minimum cost rewrite $r^*$ of $W$.

Our rewrite algorithm considers views in $V$ during the search for $r^*$. Since some views may contain UDFs, for the rewriter to utilize those views during its search, some understanding of UDFs is required. Next we will describe our UDF model and then present our rewrite algorithm that solves this problem.

3. UDF MODEL

Since big data queries frequently include UDFs, in order to reuse previous computation in our system effectively we desire a way to model MR UDFs semantically. If the system has no semantic understanding of the UDFs, then the opportunities for reuse will be limited — essentially the system will only be able to exploit cached results when one query applies the exact same UDF to the exact same input as a previous query. However, to the extent that we are able to “look into” the UDFs and understand their semantics, there will be more possibilities for reusing previous results. In this section we propose a UDF model that allows a deeper semantic understanding of MR UDFs. Our model is general enough to capture a large class of UDFs that includes classifiers, NLP operations (e.g., taggers, sentiment), text processors, social network (e.g., network influence, centrality) and spatial (e.g., nearest restaurant) operators. As an example, we performed an empirical analysis of two real-world UDF libraries, Piggybank [22] and DataFu [5]. Our model captures about 90% of the UDFs examined: 16 out of 16 Piggybank UDFs, and 30 out of 35 DataFu UDFs as detailed in [17]. Of course, we do not require the developer to restrict herself to this model; rather, to the extent a query uses UDFs that follow this model, the opportunities for reuse will be increased.

3.1 Modeling a UDF

![Figure 2: A UDF composed of local functions ($lf_1$, $lf_2$, · · · , $lf_k$), showing the end-to-end transformation of input to output.](image-url)

We propose a model for UDFs that allows the system to capture a UDF as a composition of local functions as shown in Figure 2, where each local function represents a map or reduce task. The nature of the MR framework is that map-reduce functions are stateless and only operate on subsets of the input, i.e., a single tuple or a single group of tuples. Hence, we refer to these map-reduce functions as local func-
tions. A local function can only perform a combination of the follow-
ing three types of operations performed by map and reduce tasks.
1. Discard or add attributes, where an added attribute and its values
   may be determined by arbitrary user code
2. Discard tuples by applying filters, where the filter predicates may
   be performed by arbitrary user code
3. Perform grouping of tuples on a common key, where the grouping
   operation may be performed by arbitrary user code

The end-to-end transformation of a UDF is obtained by composites
operations performed by each local function f in the UDF. Our model captures the fine-grain dependencies between the input and
output tuples in the following way.

The UDF input is modeled as \((A, F, K)\) where \(A\) is the set of
attributes, \(F\) is set of filters previously applied to the input, and \(K\)
is the current grouping of the input, which captures the keys of the
data. The output is modeled as \((A', F', K')\) with the same semantics.
Our model describes a UDF as the transformation from \((A, F, K)\) to
\((A', F', K')\) as performed by a composition of local functions using
operation types (1) (2) (3) above. Figure 2 shows how to semantically model a UDF that takes any arbitrary input represented as \(A, F, K\)
and applies local functions to produce an output that is represented as
\((A', F', K')\). Additionally, for any new attribute produced by a UDF
(in the output schema \(A'\)), its dependencies on the input (in terms of
\(A, F, K\)) are recorded as a signature along with the unique UDF-
name. Note that since the model only captures end-to-end transfor-
mations, for a UDF containing multiple internal jobs (e.g., the local
functions in Figure 2), the system only retains the final output but not
the intermediate results of local functions.

The model also captures UDFs that take multiple inputs, which is
similar to the single input case shown in Figure 2. For example, a
UDF that combines 2 inputs on a common key (similar to an equi-
join) can be described in the following way. The inputs \((A_1, F_1, K_1)\)
and \((A_2, F_2, K_2)\) produce an output \((A_J, F_J, K_J)\) such that: \(A_J\)
is the union of \(A_1\) and \(A_2\), \(F_J\) is the conjunction of the filters \(F_1, F_2\)
and the join condition, and \(K_J\) is the union of \(K_1\) and \(K_2\) intersec-
ted with the join attributes. Note that the model only concerns itself with the end-to-end transformation of the inputs to the outputs, the actual implementation of an operator is not captured by the model.

As an example, consider UDF_FOODIES that applies a food senti-
ment classifier on tweets to identify users that tweet positively about
food. An abbreviated HiveQL definition of the UDF is given in Fig-
ure 3(a) that computes a sentiment value for each tweet about food. If
a threshold operator, as noted above in (2) the filter expression may be
a function containing arbitrary user code (i.e., a UDF) and may even
contain a nesting of UDFs.

The two local functions correspond to arbitrary user code that per-
form complex text processing tasks such as parsing, word-stemming,
entity tagging, and word sentiment scoring. Yet, the UDF model succin-
tly captures the end-to-end transformation of this complex UDF
as shown in Figure 3(b). In the figure, the end-to-end transformation of
UDF_FOODIES is captured by recording the changes made to the
input \(A, F\) and \(K\) by the UDF functions that produces \(A', F'\) and \(K'\)
using a simple notation. Furthermore, for the new attribute sent_sum
in \(A'\), its dependencies on the subset of the inputs are recorded.
We provide a more concrete example of the application of the UDF
model in a HiveQL query in Section 3.2. In this way, the model en-
codes arbitrary user-code representing a sequence of MR jobs, by only
capturing its end-to-end transformations.

Our approach represents a gray-box model for UDFs, giving the
system a limited view of the UDF’s functionality yet allowing the sys-
tem to understand the UDF’s transformations in a useful way. In con-
trast, a white-box approach requires a complete understanding of how
the transformations are performed, imposing significant overhead on
the system. While with a black-box model, there is very little overhead
but no semantic understanding of the transformations, limiting the
opportunity to reuse any previous results.

### 3.2 Applying the UDF Model and Annotations

Having presented our model for UDFs, we now show how to use it to annotate a query plan that contains both UDFs and relational op-
erators. In Figure 4(a), we show a query that uses Twitter data to
identify prolific users who talk positively about food (i.e., “foodies”).
The query is expressed in a simplified representation of HiveQL and ap-
pplies UDF_FOODIES from Figure 3(a) that computes a food senti-
ment score (sent_sum) per user based on each user’s tweets.

The HiveQL query is converted to an annotated plan as shown in
Figure 4(b) by utilizing the UDF model of UDF_FOODIES as given in
Figure 3(b). In addition to modeling UDFs, the three operations types
denoted as (1, 2, 3 above) can also be used to characterize standard
relational operators such as select (2), project (1), join (2,3), group-

---

**Figure 3: UDF_FOODIES a) implementation composed of two lo-
cal functions, b) UDF model showing the end-to-end transforma-
tion of input to output.**

As an example, consider UDF_FOODIES that applies a food senti-
ment classifier on tweets to identify users that tweet positively about
food. An abbreviated HiveQL definition of the UDF is given in Fig-
ure 3(a) that invokes the following two local functions If1 and If2
written in a high-level language (Perl in this example). If1: For each
(user_id, tweet_text), apply the food sentiment classifier function
that computes a sentiment value for each tweet about food. If2: For
each user_id, compute the sum of the sentiment values to pro-
duce sent_sum, then filter out users with a total score greater than
a threshold. Although the filter in this example is a simple com-

---

**Figure 4: (a) Example query to obtain prolific foodies, and (b) cor-
responding annotated query plan.**
by (3), and aggregation (3.1). Joins in MR can be performed as a 
grouping of multiple relations on a common key (e.g., co-group in 
Pig) and applying a filter. Similarly, aggregations are a re-keying of 
the input (reflected in \( K' \)) producing a new output attribute (reflected 
in \( A' \)). These \( A, F, K \) annotations can be applied to both UDFs and 
relational operations, enabling the system to automatically annotate 
every edge in the query plan.

Figure 4(b) shows the input to the UDF is modeled as 
\( \{A\} = \{\text{user\_id}, \text{tweet\_id}, \text{tweet\_text}\} \), \( F = \emptyset \), \( K = \text{tweet\_id} \). The output is 
\( \{A'\} = \{\text{user\_id}, \text{sent\_sum}\} \), \( F' = \text{sent\_sum} \). \( K' = \text{user\_id} \).
UDF\_FOODIES produces the new 
attribute \( \text{sent\_sum} \) whose dependencies are recorded (i.e., signature) as: 
\( \{A\} = \{\text{user\_id}, \text{tweet\_text}\} \), \( F = \emptyset \), \( K = \text{tweet\_id} \), \( \text{udf\_name} = \text{UDF\_FOODIES} \). Lastly, as shown in Figure 4(b), the output 
of the UDF \( \{A', F', K'\} \) forms one input to the subsequent join oper-
ator, which in turn transforms its inputs to the final result.

This example shows how a query containing a UDF with arbitrary 
user code can be semantically modeled. The \( A, F, K \) properties are 
straightforward and can be provided as annotations by the UDF cre-
ator with minimal overhead, or alternatively they may be automati-
cally deduced using a static code analysis method such as [13], which 
is an emerging area of research. In this paper, we rely on the UDF 
creator to provide annotations, which is a one-time effort. From our 
experience evaluating the two UDF libraries [5, 22] it took less than 10 
minutes per UDF to examine the code and determine the annotations.

As noted earlier, our model is expressive enough to capture a large 
class of common UDFs. Two classes of UDFs not captured by our 
model, as noted in [17], are: (a) non-deterministic UDFs such as those 
that rely on runtime properties (e.g., current time, random, and stateful 
UDFs) and (b) UDFs where the output schema itself is dependent upon 
the input data values (e.g., pivot UDFs, contextual UDFs).

### 4. USING THE UDF MODEL TO PERFORM 
REWRITES

Our goal is to leverage previously computed results when answering 
a new query. The UDF model aids us in achieving this goal in three 
ways, as described in the following three sections. First, it provides 
a way to check for equivalence between a query and a view. Second, 
it aids in the costing of UDFs. Third, it provides a lower-bound on the 
cost of a potential rewrite.

#### 4.1 Equivalence Testing

The system searches for rewrites using existing views and can test 
for semantic equivalence in terms of our model. We consider a query 
and a view to be equivalent if they have identical \( A, F \) and \( K \) prop-
erties. If a query and a view are not equivalent, our system considers 
applying transformations (sometimes referred to as compensations) to 
make the existing view equivalent to the query.

Here we develop the mechanics to test if a query \( q \) (i.e., a target in 
the annotated plan) can be rewritten using an existing view \( v \). Query 
\( q \) can be rewritten using view \( v \) if \( v \) contains \( q \). The containment 
problem is known to be computationally hard [2] even for the class of 
conjunctive queries, hence we make a first guess that only serves as a 
quick conservative approximation of containment. This conservative 
guess allows us to focus computational efforts toward checking con-
tainment on the most promising previous results and avoid wasting 
computational effort on less promising ones.

We provide a function \text{GUSSCOMPLETE}(q, v) that performs this 
heuristic check. \text{GUSSCOMPLETE}(q, v) takes an optimistic ap-
proach, representing a \( q \) that \( v \) can produce a complete rewrite of 
\( q \). This guess requires the following necessary conditions as described 
in [10] (SPI) and [7] (SPJGA) that a view must satisfy to participate 
in a complete rewrite of \( q \).

(i) \( v \) contains all attributes required by \( q \); or contains all necessary 
attributes to produce those attributes in \( q \) that are not in \( v \)

(ii) \( v \) contains weaker selection predicates than \( q \)

(iii) \( v \) is less aggregated than \( q \)

The function \text{GUSSCOMPLETE}(q, v) performs these checks and 
returns true if \( v \) satisfies the properties i–iii with respect to \( q \). Note 
these conditions under-specify the requirements for determining that a 
valid rewrite exists, as they are necessary but not sufficient conditions. 
Thus the guess may result in a false positive, but will never result in a 
false negative. The purpose of \text{GUSSCOMPLETE}(q, v) is to provide 
a quick way to distinguish between views that can possibly produce a 
rewrite from views that cannot. As rewriting is an expensive process, 
this helps to avoid examining views that cannot produce valid rewrites.

#### 4.2 Costing a UDF

Given that our goal is to find a low cost rewrite for queries contain-
ing UDFs, we require a method of costing an MR UDF. We define the 
cost of a UDF as the sum of the cost of its local functions. Estimating 
the cost of a local function that performs any of the three operation 
types is complicated by two factors:

(a) Each operation type is performed by arbitrary user code, and thus 
can have varying complexity. For instance, although an NLP sentence 
tagger and a simple word-counter function perform the same 
operation type (discard or add attributes), they can have signifi-
cantly different computational costs.

(b) There could be multiple operation types performed in the same 
local function, making it unrealistic to develop a cost model for 
every possible local function.

Due to these factors, we desire a conservative way to estimate the 
cost of a local function of varying complexity that may apply a se-
quence of operation types without knowing specifically how these op-
erations interact with each other inside the local function.

Developing an accurate cost model is a general problem for any 
database system. In our framework, the importance of the cost model 
only in guiding the exploration of the space of rewrites. For this 
reason, we appeal to an existing cost model from the literature [21], 
but slightly modify it to be able to cost UDFs. To this end, we extend 
the “data only” cost model in [21] in a limited way so that we are able 
to produce cost estimates for UDFs. Although this results in a rough 
cost estimate, experimentally we show that our cost model is effective 
in producing low cost rewrites (Section 8). The cost model we develop 
here is simple but works well in practice; however, an improved cost 
model may be plugged-in as it becomes available.

Recall that UDFs are composed of local functions, where each local 
function must be performed by a map task or a reduce task. The cost 
model in [21] accounts for the “data” costs (read/write/shuffle), and we 
augment it in a limited way to account for the “computational” 
cost of local functions. Since a UDF can encompass multiple jobs, 
we express the cost of each job as the sum of: the cost to read the 
data and apply a map task (\( C_m \)), the cost of sorting and copying (\( C_s \)), 
the cost to transfer data (\( C_t \)), the cost to aggregate data and apply a 
reduce task (\( C_r \)), and finally the cost to materialize the output (\( C_w \)). 
Using this as a \textit{generic} cost model, we first describe our approach 
toward solving (a) above by assuming that each local function only 
performs one instance of a single operation type. Then we describe 
our approach for (b), above.

For (a) we model the cost of the three operation types rather than 
each local function, which provides the baseline cost value for each 
operation type. Since there may be a high variation in the cost of a 
UDF’s local functions, we apply a scalar multiplier to the baseline 
cost of \( C_m, C_r \). To calibrate \( C_m, C_r \) we take an empirical approach 
to estimate the scalar values. The first time the UDF is added to the
system, we execute the UDF on a 1% uniform random sample of the input data to determine the scalar values. Due to data skew and one-time calibration, this may result in imprecise cost estimates. However, we do not preclude (a) recalibrating $C_m, C_v$ when the UDF is applied to new data, (b) a better sampling method if more is known about the data, and (c) periodically updating $C_m, C_v$ after executing the UDF on the full dataset.

For (b), since a local function performs an arbitrary sequence of operations of any type, it is difficult to estimate its cost. This would require knowing how the different operations actually interact with one another, which requires a white-box approach. For this reason we desire a conservative way to estimate the cost of a local function, which we do by appealing to the following property of any cost model performing a set $S$ of operations.

**Definition 1. Non-subsumable cost property:** Let $\text{Cost}(S, D)$ be defined as the total cost of performing all operations in $S$ on a database instance $D$. The cost of performing $S$ on $D$ is at least as much as performing the cheapest operation in $S$ on $D$.

$$\text{Cost}(S, D) \geq \min(\text{Cost}(x, D), \forall x \in S)$$

The gray-box model of the UDFs only captures enough information about the local functions to provide a cost corresponding to the least expensive operation performed on the input. We cannot use the most expensive operation in $S$ (i.e., $\max(\text{Cost}(x, D), \forall x \in S)$), since this requires $\text{Cost}(S', D) \leq \text{Cost}(S, D)$, where $S' \subseteq S$. The “max” requirement is difficult to meet in practice, which we can show using a simple example. Suppose $S$ contains a filter with high selectivity, and a group-by with higher cost than the filter when considering these operations independently on database $D$. Let $S'$ contain only group-by. Suppose that applying the filter before group-by results in few or no tuples streamed to group-by. Then applying group-by can have nearly zero cost and it is plausible that $\text{Cost}(S', D) > \text{Cost}(S, D)$.

The cost model utilizes the non-subsumable cost property in the following way. A local function that performs multiple operation types $t$ is given an initial cost corresponding to the generic cost of applying the cheapest operation type in $t$ on its input data. This initial value can then be scaled-up as described previously in our solution for (a).

### 4.3 Lower-bound on Cost of a Potential Rewrite

Now that we have a quick way to determine if a view $v$ can potentially produce a rewrite for query $q$, and a method for costing UDFs, we would like to compute a quick lower bound on the cost of any potential rewrite – without having to actually find a valid rewrite, which is computationally hard. To do this, we will utilize our UDF model and the non-subsumable cost property when computing the lower-bound. The ability to quickly compute a lower-bound is a key feature of our approach.

$$A = \{a,b,c\} \quad F = \{\} \quad K = \{\}$$

We define an optimistic cost function $\text{OptCost}(q, v)$ that computes this lower-bound on any rewrite $r$ of query $q$ using view $v$ only if $\text{GuessComplete}(q, v)$ is true. Otherwise $v$ is given $\text{OptCost}$ of $\infty$, since in this case it cannot produce a complete rewrite, and hence the $\text{Cost}$ is also $\infty$. The properties of $\text{OptCost}(q, v)$ are that it is very quick to compute and

$$\text{OptCost}(q, v) \leq \text{Cost}(r).$$

When searching for the optimal rewrite $r^*$ of $W$, we use $\text{OptCost}$ to enumerate the space of the candidate views based on their cost potential, as we describe in the next section. This is inspired by nearest neighbor finding problems in metric spaces where computing distances between objects can be computationally expensive, thus preferring an alternate distance function (e.g., $\text{OptCost}$) that is easy to compute with the desirable property that it is always less than or equal to the actual distance.

### 5. Problem Overview for Rewriting Queries Containing UDFs

Our UDF model enables reuse of views to improve query performance even when queries contain complex functions. However, reusing an existing view when rewriting a query with any arbitrary UDF requires the rewrite process to consider all UDFs in the system. The rewrite problem is known to be hard even when both the queries and the views are expressed in a language that only includes conjunctive queries [10, 19].

In our scenario, users are likely to include many UDFs in their queries. If the rewrite process were to consider every UDF as an operator in the rewrite language, searching for the optimal rewrite would quickly become impractical for any realistic workload and number of views. This is because the search space for finding a rewrite is exponential in both 1) the number of views in $V'$ and 2) the number of operators (e.g., Relational and UDFs) considered by the rewrite process, which may include multiple applications of the same operator. For our rewrite algorithm, the worst case complexity is $O(n \cdot |V| \cdot D \cdot |k|)$ where $n$ is the number of nodes in the plan $W$, $J$ is the maximum number of views that can participate in a rewrite, $|V|$ represents the number of views in the system, $k$ is maximum number of times that a particular operator can appear in a rewrite, and $|I_k|$ is the number of operators considered by the rewrite algorithm. For the experimental evaluation of our rewrite algorithm presented in Section 8, we set $J = 4$ and $k = 2$ for practical reasons.
In our system, both the queries and the views can contain any arbitrary UDF, creating a potentially large number of UDFs in the system. Due to the complexity of the rewrite search process, in practice it is a good idea to limit the rewrite process to consider only a small subset of all UDFs in the system. For this reason, in our system the rewriter considers relational operators — select, project, join, group-by, aggregations (SPJGA), and a few of the most frequently used UDFs, which increases the possibility of reusing previous results. Selecting the right subset of UDFs to include in the rewrite process is an interesting open problem that must consider the tradeoff between the added expressiveness of the rewrite process versus the additional exponential cost incurred to search for rewrites.

A naive solution is to search for the optimal rewrite only for target \( W_n \). However, (a) even if a rewrite is found for \( W_n \), there may be a cheaper rewrite of \( W \) using a rewrite found for a different target \( W_i \), and (b) if one cannot find a rewrite for \( W_n \), one may be able to find a rewrite at a different target \( W_i \). The source of this problem is that \( W_n \) may contain a UDF that is not included in the set of rewrite operators, and hence search process cannot be restricted only to \( W_n \). For example, a rewrite for \( W_n \) can be expressed by composing a rewrite \( r_i \) for a target \( W_i \) with the remaining nodes in \( W \) indicated by \( \text{NODE}_{i+1} \cdots \text{NODE}_n \). The composition of this rewrite could be cheaper than the rewrite found at \( W_n \), thus the search process for the optimal rewrite must happen at all \( n \) targets in \( W \).

A better solution is to independently search for the best rewrite at each of the \( n \) targets of \( W \), and then use a dynamic programming solution to choose a subset among these to obtain the optimal rewrite \( r^\ast \). One drawback of this approach is that there is no way of early terminating the search at a particular target since each search is independent. Hence, the search at one target does not inform the search at another. For instance, the algorithm may have searched for a long time at a target \( W_i \) only to find an expensive rewrite, when it could have found a better (lower-cost) rewrite at an upstream target \( W_{i-1} \) more quickly had it known to look there first.

The approach we take in this paper, called B\textsc{Rewrite}, remedies these two shortcomings of the dynamic programming approach by (1) using the lower bound function OPT\textsc{Cost} introduced in Section 4.3 to guide the search process at each target, and (2) using results from the search process at one target to guide the search at the other targets. First, after finding a rewrite \( r \) with cost \( c \) at a target \( W_i \), there is no need to continue searching for rewrites at \( W_i \) if the OPT\textsc{Cost} of the the remaining unexplored space at \( W_i \) is greater than \( c \). Second, \( r \) and \( c \) can be used to prune the search space at other targets in \( W \) by composing a rewrite of \( W_n \) using \( r \) and the remaining nodes (e.g., \( \text{NODE}_{i+1} \cdots \text{NODE}_n \)) in \( W \).

![Figure 6: High level overview of the B\textsc{Rewrite} algorithm.](image)

Figure 6 provides a high-level overview of our B\textsc{Rewrite} algorithm with plan \( W \) represented as a DAG. Each node is associated with an instance of the VIEW\textsc{Finder} module (VF), which is represented as a black box alongside described below. Additionally, each node stores its best rewrite found so far along with its cost. B\textsc{Rewrite} interacts with this DAG using a function that identifies the next target to continue the rewrite search. On the right side of the figure, the interface to the black box VIEW\textsc{Finder} is shown, which implements 3 simple primitives. Using this setup, the B\textsc{Rewrite} algorithm performs a search for the globally optimal rewrite of \( W \). There are 3 main components to the B\textsc{Rewrite} algorithm.

1. **VIEW\textsc{Finder}** at each target implements three operations — INIT sets up the initial search space of candidate views, ordering the available views by their OPT\textsc{Cost}; PEAK provides the OPT\textsc{Cost} of the next potential rewrite at the target; and REFINE which incrementally grows the space and attempts to find a rewrite of the target. This constitutes the local search at each target.

2. **B\textsc{Rewrite}’s FIND\textsc{NextMinTarget}** interface queries the DAG to identify the next target to explore. This constitutes the global search among the targets in \( W \).

3. When a low-cost rewrite is found at a target, it is propagated to the remaining targets in \( W \) by updating their best plan and its cost (BEST\textsc{Plan} and BEST\textsc{PlanCost}). This constitutes the update mechanism that coordinates the search of all targets in \( W \).

These three components represent the global logic of B\textsc{Rewrite} that explores the rewrite search space, at each step deciding the next target to explore. For each local search, the termination condition is that the remaining views to be examined (PEAK) have a lower bound cost that is greater than the best rewrite found so far. For the global search, the termination condition is that none of the targets has a potential of producing a lower cost rewrite of \( W_i \) than the best one found so far. Note that due to the propagation, a node’s best plan and cost do not necessarily correspond to the best rewrite found by the VIEW\textsc{Finder} at that particular node, but could be a composition of rewrites found at other nodes.

In the next two sections, we provide the details of these components and the process outlined above. We first describe B\textsc{Rewrite} in Section 6.1, which is the main driver of the rewrite search process, and is shown in Algorithm 1 and Algorithm 2. The mechanism to propagate the best rewrite is given in Algorithm 3. Then in Section 7 we describe the details of the VIEW\textsc{Finder} component that is utilized as black box in the figure above.

### 6. BEST-FIRST REWRITE

The B\textsc{Rewrite} algorithm produces a rewrite of \( W \) that can be composed of rewrites found at multiple targets in \( W \). The computed rewrite \( r^\ast \) has provably the minimum cost among all possible rewrites in the same class. Moreover, the algorithm is work-efficient: even though \( \text{Cost} (r^\ast) \) is not known a-priori, it will never examine any candidate view with OPT\textsc{Cost} higher than the optimal cost \( \text{Cost} (r^\ast) \). To be work efficient, the algorithm must choose wisely the next candidate view to examine. As we will show below, the OPT\textsc{Cost} functionality plays an essential role in choosing the next target to refine. Intuitively, the algorithm explores only the part of the search space that is needed to provably find the optimal rewrite. We prove that B\textsc{Rewrite} finds \( r^\ast \) while being work-efficient in Section 6.2.

#### 6.1 The B\textsc{Rewrite} Algorithm

Algorithm 1 presents the main B\textsc{Rewrite} function. In lines 2–6, B\textsc{Rewrite} initializes a VIEW\textsc{Finder} at each target \( W_i \) and sets BEST\textsc{Plan} and BEST\textsc{PlanCost} to be the original plan and its cost. In lines 7–10, it repeats the following procedure: Invoke FIND\textsc{NextMinTarget} (described in Algorithm 2) to choose the next best target to continue the search, which returns \( (W_i, d) \), indicating that target \( W_i \) can potentially produce a rewrite with a lower bound cost of \( d \). Next, invoke REFINE\textsc{Target} (described in Algorithm 2) which asks the VIEW\textsc{Finder} to search for the next rewrite at target \( W_i \). This continues until there is no target that can possibly improve BEST\textsc{Plan}_n, at which point BEST\textsc{Plan}_n (i.e., \( r^\ast \)) is returned.

FIND\textsc{NextMinTarget} in Algorithm 2 identifies the next best target \( W_i \) to be refined in \( W \), as well as the minimum cost (OPT\textsc{Cost})
of a potential rewrite for \( W_i \). There can be three outcomes of a search at a target \( W_i \): Case 1: \( W_i \) and all its ancestors cannot provide a better rewrite. Case 2: An ancestor target of \( W_i \) can provide a better rewrite. Case 3: \( W_i \) can provide a better rewrite. By recursively making the above determination at each target \( W_i \) in \( W \), the algorithm identifies the best target to refine next.

For a target \( W_i \), the cost \( d' \) of the cheapest potential rewrite that can be produced by the ancestors of \( \text{NODE}_i \) is obtained by summing the \text{VIEWFINDER.PEEK} values at \( \text{NODE}_i \)'s ancestors nodes and the cost of \( \text{NODE}_i \) (lines 3–11). Note that we also record the target \( W_{MIN} \) representing the ancestor target with the minimum \text{OptCost} candidate view (lines 6–9). Then \( d_i \) is assigned to the next candidate view at \( W_i \), using \text{VIEWFINDER.PEEK} (line 12).

Next the algorithm deals with the three cases outlined above. If both \( d' \) and \( d_i \) are greater than or equal to \( W_{MIN} \), the candidate is assigned to the next candidate view at \( W_i \) (line 13). Otherwise, \( W_{MIN} \) is the next target to refine (case 2). Else (line 18), \( W_i \) is the next target to refine (case 3).

Finally, \text{REFINETARGET} in Algorithm 2 describes the process of refining a target \( W_i \). Refinement is a two-step process. In the first step it obtains a rewrite \( r_i \) of \( W_i \) from \text{VIEWFINDER} if one exists (line 2). The cost of the rewrite \( r_i \) obtained by \text{REFINETARGET} is compared against the best rewrite found so far at \( W_i \). If \( r_i \) is found to be cheaper, the algorithm suitably updates \( \text{BESTPLAN}_i \) and \( \text{BESTPLANCost}_i \) (lines 3–9). In the second step (line 7), the algorithm tries to compose a new rewrite of \( W_i \) using \( r_i \), through the recursive function given by \text{PROPBESTWRITE} in Algorithm 3. After this two-step refinement process, \( \text{BESTPLAN}_i \) contains the best rewrite of \( W_i \) found so far.

\text{PROPBESTWRITE} in Algorithm 3 describes the recursive update mechanism that pushes the new \( \text{BESTPLAN}_i \) downward along the outgoing nodes and towards \( \text{NODE}_i \). At each step it composes a rewrite \( r_i \), using the immediate ancestor nodes of \( \text{NODE}_i \) (lines 2–5). It compares \( r_i \) with \( \text{BESTPLAN}_i \), and updates \( \text{BESTPLAN}_i \) if \( r_i \) is found to be cheaper (lines 6–12).

### 6.2 Proof of Correctness and Work-Efficiency

The following theorem provides the proof of correctness and the work-efficiency property of our \text{BFREWRITE} algorithm.

**Theorem 1.** \text{BFREWRITE} finds the optimal rewrite \( r^* \) of \( W \) and is work-efficient.

**Proof.** To ensure correctness, finding the optimal rewrite requires that the algorithm must not terminate before finding \( r^* \). To ensure work-efficiency (defined earlier) requires that the algorithm should not examine any candidate views that cannot be possibly included in \( r^* \).

A proof sketch by contradiction for a single target case (i.e., \( n = 1 \)) is as follows. Assume two cases: First, suppose that the algorithm found a candidate view \( v \) resulting in a rewrite \( r \), while the candidate view \( v' \), which produces the optimal rewrite \( r^* \), is not considered before terminating even though \( \text{Cost}(r) > \text{Cost}(r^*) \). Second, the algorithm examined a candidate view \( v' \) with \( \text{OptCost}(v') > \text{cost}(r^*) \). We can then show both these cases are not possible, proving that \text{BFREWRITE} finds \( r^* \) in a work-efficient manner. The full proof for this single target case, which is then extended to the multi-target case, is provided in the extended version [17].

### 7. VIEWFINDER

The key feature of \text{VIEWFINDER} is its \text{OptCost} functionality that enables it to incrementally explore the space of rewrites using the views in \( W \). As noted earlier in Section 4.1, rewriting queries using views is known to be a hard problem. Traditionally, methods for rewriting queries using views for SPJG queries use a two stage approach [1, 10]. The pruning stage determines which views are relevant to the query, and among the relevant views, those that contain all the required join predicates are termed as complete otherwise they are called partial solutions. This is typically followed by a merge stage that joins the partial solutions using all possible equijoin methods to form additional relevant views. The algorithm repeats until only those views that are useful for answering the query remain.

We take a similar approach in that we identify partial and complete solutions, then follow with a merge phase. The \text{VIEWFINDER} considers candidate views \( C \) when searching for rewrite of a target. \( C \) includes views in \( V \) as well as views formed by “merging” views in
V using a MERGE function, which is an implementation of a standard view-merging procedure (e.g., [1, 10]). Traditional approaches begin merging partial solutions to create complete solutions, until no partial solutions remain. This “explodes” the space of candidate views exponentially up-front. In contrast, our approach gradually explodes the space, resulting in far fewer candidates views from being considered.

Additionally, with no early termination condition, existing approaches would need to explore the space exhaustively at all targets. The VIEWFINDER incrementally grows and explores only as much of the space as needed, frequently stopping and resuming the search as requested by BfRewrite.

7.1 The VIEWFINDER Algorithm

The VIEWFINDER is presented in Algorithm 4. An instance of VIEWFINDER instantiated at each target, which is stateful; enabling it to start, stop, and resume the incremental searches at each target. The VIEWFINDER maintains state using a priority queue (PQ) of candidate views, ordered by OptCost. VIEWFINDER implements the Init, Peek, and Refine functions.

The Init function instantiates a VIEWFINDER with a query q representing a target W, and all views in V are added to PQ, which orders them by increasing OptCost. The Peek function returns the head item in PQ. The Refine function is invoked when BfRewrite asks the VIEWFINDER to examine the next candidate view.

Refine pops the head item v out of PQ and generates a set of new candidate views M by merging v with those views previously popped from PQ which were stored in Seen. Note that Seen only contains candidate views that have an OptCost less than or equal to that of v. Critically, this results in an “on-demand” incremental growth of the candidate space as required by BfRewrite, rather than performing a pre-explosion of the entire search space. A property of the new candidate views in M, which is required for the correctness of the algorithm, is that they have an OptCost greater than v, hence none of these views could have been examined before v. This property is provided as a theorem in the extended version [17]. All newly created views in M are inserted into PQ and v is then added to Seen.

The Refine function next attempts to find a rewrite using view v by invoking RewriteEnum, described next. Given the computational complexity of finding valid rewrites, VIEWFINDER limits the invocation of the RewriteEnum algorithm using two strategies. First, the expensive RewriteEnum operation is only applied to the view at the head of PQ when requested by BfRewrite. Second, it avoids applying RewriteEnum on every candidate view unless it passes the GuessComplete test as described in Section 4.3.

7.2 Rewrite Enumeration

The RewriteEnum function searches for a valid rewrite of query q using view v that has passed the GuessComplete test. Since GuessComplete can result in false positives, there is no guarantee that v will produce a valid rewrite for q. However, if a rewrite exists, RewriteEnum returns the rewrite and its cost as computed by the Cost function.

In searching for a rewrite, recall from Section 5 that the rewrite process considers relational operators SPIGAs and a subset of the UDFs in the system. These are the only rewrite operators considered by RewriteEnum. The rewrite process searches for equivalent rewrites of q by applying compensations [29] to v and then testing for equivalence against q. In our implementation of RewriteEnum this is done by generating all permutations of the rewrite operators and testing for equivalence, amounting to a brute force enumeration of all possible rewrites that can be produced with compensations. This makes the case for the system to keep the set of rewrite operators small since this search process is exponential in the size of this set. However, when the rewrite operators are restricted to a fixed known set, it may suffice to examine a polynomial number of rewrite attempts as in [8] for

Algorithm 4 VIEWFINDER

```
1: function Init(q, v)
2:     Priority Queue PQ ← ∅; Seen ← ∅; Query q
3:     q ← query
4:     for each v ∈ V do
5:         PQ.add(v, OptCost(q, v))
6:     end for
7: end function

1: function Peek
2:     if PQ is not empty return PQ.peek(), OptCost else ∞
3: end function

1: function Refine
2:     if not PQ.empty() then
3:         v ← PQ.pop()
4:         M ← Merge(v, Seen) > Discard from M those in Seen ∩ M
5:         for each v′ ∈ M do
6:             PQ.add(v′, OptCost(q, v′))
7:         end for
8:         Seen.add(v)
9:     if GuessComplete(q, v) then
10:        return RewriteEnum(q, v)
11:     end if
12:     end if
13:     return NULL
14: end function
```

the specific case of simple aggregations involving group-bys. Such approaches are not applicable to our case as the system has the flexibility to add any UDF to the set of rewrite operators.

8. EXPERIMENTAL EVALUATION

In this section, we present an experimental study showing the effectiveness of BfRewrite in finding low-cost rewrites of complex queries. First, we evaluate our methods in two scenarios. The query evolution scenario (Section 8.3.1) represents a user iteratively refining a query within a single session. This scenario evaluates the benefit that each new query version can receive from the opportunistic views created by previous versions of the query. The user evolution scenario (Section 8.3.2) represents a new user entering the system presenting a new query. This scenario evaluates the benefit a new query can receive from the opportunistic views previously created by queries of other "similar" users. Next, we evaluate the scalability (Section 8.3.3) of our rewrite algorithm in comparison to a competing approach. Lastly, we compare our method to cache-based methods (Section 8.3.4) that can only reuse previous results when they are identical.

8.1 Query Workload

We first provide some insights into the characteristics of exploratory processing on big data and then describe the workload from [16] that we adopted for use in this paper. Recent work [3, 4, 24] examines MR queries “in the wild” on production and research clusters that were utilized by data scientists or other advanced users for up to a year. Additional work [11, 14, 27, 28] provides further insights into big data analytical queries. A key finding of [4] is that there is a need for better benchmarks to capture the use cases for MR queries that perform interactive analysis on big data. Below we summarize the main findings from our literature review.

1. Users spend time revising and improving exploratory queries [14, 24], and thus queries near the end of an exploratory session tend to represent higher-quality and more complex versions of earlier queries [14].
2. Many studies note that complex analysis on big data frequently include UDFs [11, 27, 28].
3. Queries frequently incorporate multiple datasets [24] with a majority of queries (65% in [24]) accessing three or more.
Both [24] and [3] note that users frequently re-access their data and there can be significant benefits from caching. A majority of jobs involve data re-accesses with many occurring within 1 hour (50% [3] and up to 90% [24]). These frequent re-access patterns make a strong case for a method such as BrRewrite.

The experimental workload from [16] contains 32 queries on three datasets that simulate 8 analysts $A_1$-$A_8$ who write complex exploratory analytical queries for business marketing scenarios. Each query uses at least one of 10 unique UDFs. The workload uses three real-world datasets: A Twitter log (TWTR) of user tweets, a FourSquare log (4SQ) of user check-ins, and a Landmarks log (LAND) containing locations of interest. Many queries begin by accessing only one or two datasets, but subsequent revisions use all three datasets. Each of the 8 analysts poses 4 versions of a query, representing the initial query followed by three subsequent revisions made during data exploration and hypothesis testing. Hence, there is some overlap expected between subsequent versions of a query. The queries are long-running with many operations, and executing the queries with Hive created 17 opportunistic materialized views per query on average.

Since each query in the workload has multiple versions, we use $A_{ij}$ to denote Analyst $i$ executing version $j$ of her query. Since there are 4 versions of each query, $A_{ij}$+$1$ represents a revision of $A_{ij}$. Below is a high-level description of query $A_{1v1}$ and $A_{1v2}$, taken from [16].

**Example 1.** Analyst $i$ ($A_i$) wants to identify a number of “wine lovers” to send them a coupon for a new wine being introduced in a local region.

Query $A_{1v1}$: (a) From TWTR, apply UDF-CLASSIFY-WINE-SCORE on each user’s tweets and group-by user to produce a wine-sentiment-score for each user and then threshold on wine-sentiment-score. (b) From TWTR, compute all pairs $(u_1, u_2)$ of users that communicate with each other, assigning each pair a friendship-strength-score based on the number of times they communicate and then threshold on the friendship-strength-score. (c) From TWTR, apply UDF-CLASSIFY-AFFLUENT on users and their tweets. Join results from (a), (b), (c) on user.id.

Query $A_{1v2}$: Revise the previous version by reducing the wine-sentiment-score threshold, adding new data sources (4SQ and LAND) to find the check-in counts for users that check-in to places of type wine-bar, then threshold on count, joining this result with the users found in the previous version. Queries $A_{1v3}$ and $A_{1v4}$ are similarly revised by changing the threshold parameters and requiring that a user’s friends also have a high check-in count to wine-bars.

The performance of any method that reuses results from previous queries will obviously depend on the degree of “similarity” between queries. However, choosing a meaningful metric to compute the similarity between queries in the workload from [16] was not clear. While methods such as [14] characterize query similarity in terms of query text (FROM clause, WHERE clause, etc.), we found this did not directly correspond with result reusability. We observed this effect in a microbenchmark we performed based on revising queries, and report those results in the extended version of the paper [17].

### 8.2 Experimental Methodology

Our experimental system consists of 20 machines running Hive version 0.7.1 and Hadoop version 0.20.2. Each node has the same hardware: 2 Xeon 2.4GHz CPUs (8 cores), 16 GB of RAM, and exclusive access to its own disk (2TB SATA 2012 model). We use HiveQL as the declarative query language, and Oozie as a job coordinator. The MR UDFs are implemented in Java, Perl, and Python and executed using the HiveCLI. UDFs implemented in our system include a log parser/extractor, text sentiment classifier, sentence tokenizer, lat/lon extractor, word count, restaurant menu similarity, and geographical tiling, among others. All UDFs are annotated using the model as per the example annotations given in Section 3.2. For each UDF in the workload we calibrate its cost model using the procedure described in Section 4.2. We provide an additional experiment in the extended version [17] to show that although we calibrate our cost model only the first time the UDF is added, it is able to discriminate between good plans and really bad plans for the purpose of query rewriting.

Our experiments use over 1TB of data that includes 800GB of TWTR tweets, 250GB of 4SQ check-ins, and 7GB of LAND containing 5 million landmarks. The identity of a user (user_id) is common across the TWTR and 4SQ logs, while the identity of a landmark (location_id) is common across 4SQ and LAND.

We report the following metrics for all experiments. Experiments on query execution time report both the original execution time of the query in Hive, labelled as ORIG, and the execution time of the rewritten query, labelled as REWR. The reported time for REWR includes the time to run the BrRewrite algorithm, the time to execute the rewritten query, and any time spent on statistics collection. Experiments on the runtime of rewrite algorithms report the total time used by the algorithm to find a rewrite of the original query using the views in the system. For these experiments, BFR denotes our BrRewrite rewriting algorithm, and $\text{DP}$ represents a competing approach based on dynamic programming. $\text{DP}$ does not use $\text{OPTCost}$, and searches exhaustively for rewrites at every target. $\text{DP}$ then rewrites a query by applying a dynamic programming solution to choose the best subset of rewrites found at each target. We note that both algorithms produce identical rewrites (i.e., $r^*$). The primary comparison metric for $\text{BFR}$ and $\text{DP}$ is algorithm runtime. In addition, we report results for two sec-
ondary metrics: the number of candidate views examined during the search for rewrites, and the number of valid rewrites attempted and produced during the search process. These correspond to the candidate space explored and rewrites attempted before identifying \( r^* \).

### 8.3 Experimental Results

#### 8.3.1 Query Evolution

In this experiment, for each analyst \( A_i \), query \( A_{i,v_1} \) is executed followed by query \( A_{i,v_2} \), \( A_{i,v_3} \), and \( A_{i,v_4} \), applying \( \text{BFR} \) and \( \text{DP} \) each time to rewrite the new query using the opportunistic views generated by the previous query versions. Before each Analyst \( A_i \) begins, all views are dropped from the system. This experiment creates a scenario where an analyst may benefit by reusing results from previous versions of their own query. Figure 7(a) shows the execution time of the original query (\( \text{ORIG} \)) and the rewritten query (\( \text{REWR} \)), while Figure 7(b) reports the corresponding percent improvement in execution time of \( \text{REWR} \) over \( \text{ORIG} \) for each query \( v_i \) (not shown since the percent improvement is always zero). Figure 7(b) shows that \( \text{REWR} \) provides an overall improvement of 10% to 90%; with an average improvement of 61% and up to an order of magnitude. As a concrete data point, \( A_{3,v_4} \) requires 54 minutes to execute \( \text{ORIG} \), but only 55 seconds to execute the rewritten query (\( \text{REWR} \)). \( \text{REWR} \) has much lower execution time because it is able to take advantage of the overlapping nature of the queries, e.g., version 2 has some overlap with version 1. \( \text{REWR} \) is able to reuse previous results, providing significant savings in both query execution time and data movement (read/write/shuffle) costs.

#### 8.3.2 User Evolution

In this experiment, each analyst (except one, a holdout analyst) executes the first version of their query. Then, we execute the first version of the holdout analyst’s query (e.g., \( A_{i,v_1} \) after applying \( \text{BFR} \)) to rewrite the holdout query using the opportunistic views generated by the previous queries. We then drop all views from the system and repeat using a different holdout analyst each time. This experiment creates a scenario where an analyst may benefit by reusing results from previous versions of other analysts’ queries. Figure 8(a) shows the execution time for \( \text{REWR} \) and \( \text{ORIG} \) for each different holdout analyst along the x-axis, while Figure 8(b) shows the corresponding data manipulated (read/write/shuffle) in GB. These data statistics are automatically collected and reported by Hadoop and include the amount of data read from HDFS, moved across the network, and written to HDFS. These results demonstrate that the execution time of \( \text{REWR} \) is always lower than \( \text{ORIG} \), and the data manipulated shows similar trends. The percentage improvement in execution time is given in Figure 8(c) which shows \( \text{REWR} \) results in an overall improvement of about 50%–90%. Of course, these results are workload dependent but they show that even when several analysts query the same data sets while testing different hypotheses, our approach is able to find some overlap and hence benefit from previous results.

#### 8.3.3 Algorithm Comparisons

We next test the scalability of \( \text{BFR} \) and \( \text{DP} \) by scaling up the number of views in the system from 1–1000 and report the algorithm runtime for both algorithms as they search for rewrites for one query (\( A_{3,v_1} \)). During the course of design and development of our system, we created and retained about 9,600 views; from these we discarded duplicate views as well as those views that are an exact match to the query (simply to prevent the algorithms from terminating trivially). In Figure 10, the x-axis reports the number of views in the system from 1–1000 and report the algorithm run time for both algorithms as they search for rewrites for one query (\( A_{3,v_1} \)). As an additional experiment for user evolution, we first execute a single analyst’s query (\( A_{3,v_1} \)) with no opportunistic views in the system, to create a baseline execution time. Then we “add” another analyst by executing all four versions of that analyst’s query, which creates new opportunistic views. Then we re-execute \( A_{3,v_1} \) and report the execution time improvement over the baseline, and repeat this process for the other remaining analysts. We chose \( A_{3,v_1} \) as it is a complex query that uses all three logs. Table 1 reports the execution time improvement when more opportunistic views are present in the system.

![Figure 9: Algorithm comparisons for (a) candidate views considered, (b) rewrite attempts, and (c) Algorithm runtime (log-scale).](image)

#### Table 1: Improvement in execution time of \( A_{3,v_1} \) as more analysts become present in the system.

<table>
<thead>
<tr>
<th>Analysts added</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improvement</td>
<td>0%</td>
<td>73%</td>
<td>73%</td>
<td>75%</td>
<td>89%</td>
<td>89%</td>
<td>89%</td>
</tr>
</tbody>
</table>
Finally, we show the effectiveness of OptCost in pruning the search space for BFR. Figure 11 shows the runtime behavior of BFR as it explores the space of rewrites in its search to identify the optimal rewrite. In this experiment, query $A_{1v1}$ is first executed, producing a number of views. For each subsequent query ($A_{1v2}, A_{1v3}, A_{1v4}$), BFR searches for a rewrite given the views produced by the previous queries. The x-axis reports the BFR’s elapsed run time, and the y-axis reports the percent error relative to the optimal rewrite, in terms of cost. For each query the error begins at 100% (i.e., no rewrite has been found yet) and as BFR finds rewrites, the error is reduced until it reaches zero percent. At this point BFR has identified the optimal rewrite and can terminate the search (note that this is the same rewrite identified by the exhaustive DP algorithm). During the initial “flat” period for each query, BFR is growing the space of candidate views by examining views with the lowest OptCost. Since they failed to produce a rewrite, BFR begins merging them with views that have the next lowest OptCost. This phase represents BFR incrementally growing the space of candidate views, which is in contrast to an exhaustive approach (DP) that first grows the entire space of all possible candidate views before beginning to search for rewrites. The execution time for BFR increases slightly for the subsequent queries $A_{1v2}$ and $A_{1v3}$ since the execution of each subsequent query adds more views to the system.

In Figure 11, BFR finds the first and second valid rewrites for query $A_{1v4}$ at about 0.9 seconds (indicated by the numbers 1 and 3) which reduces the error to 96% and 90% respectively. At shortly after 1 second, BFR finds valid rewrite number 46 which is the optimal rewrite and can terminate the search (note that this is the same rewrite identified by the exhaustive DP algorithm). Two notable take-aways from Figure 11 are: (a) once BFR finds the first rewrite, it quickly converges to the optimal rewrite, and (b) when BFR finds the optimal rewrite and terminates for $A_{1v4}$, it only had to find 46 rewrites before terminating, while the DP algorithm (not shown in figure) found 4656 rewrites. Similarly, BFR only had to examine $A_{1v2}$ and $A_{1v3}$ respectively, whereas DP found 66 and 323. This result illustrates a case when BFR can terminate early without examining all possible rewrites. These observations suggest that the OptCost is effective at pruning the search space for BFR.

8.3.4 Comparison with Caching-based methods

Next we provide a brief comparison of our approach with caching-based methods (such as [6]) that perform only syntactic matching when reusing previous results. With this class of solutions, results can only be reused to answer a new query when their respective execution plans are identical, i.e., the new query’s plan and the plan that produced the previous results must be syntactically identical. This means that if the respective plans differ in any way (e.g., different join orders or predicate push-downs), then reuse is not possible. For instance, with syntactic matching, a query that applies two filters in sequence $a, b$ will not match a view (i.e., a previous result) that has applied the same two filters in a different sequence $b, a$. In contrast, our BFR approach performs semantic matching and query rewriting. In this case, not only will BFR match $a, b$ with $b, a$, but it would also match the query to a view that only has filter $b$, by applying an additional filter $a$ during the rewrite process.

To represent the class of syntactic caching methods, we present a conservative variant of our approach that performs a rewrite only if a view and a query have identical $A, F, K$ properties as well as have identical plans. We term this variant BFR-SYNTACTIC. Figure 12 highlights the limitations of caching-based methods by repeating the query evolution experiment for Analyst 1 ($A_{1v1}, A_{1v4}$). We first execute query $A_{1v1}$ to produce opportunistic views, and then we apply both BFR and BFR-SYNTACTIC to queries $A_{1v2}, A_{1v3}$ and $A_{1v4}$ and report the results in terms of query execution time improvement of the solutions produced by BFR and BFR-SYNTACTIC.

Figure 12 shows that both BFR and BFR-SYNTACTIC result in the same execution time improvement for $A_{1v2}$. This is because both methods were able to reuse some of the (syntactically identical) views from the previous query. However, BFR-SYNTACTIC performs worse than BFR for query $A_{1v3}$ and $A_{1v4}$. This is because BFR-SYNTACTIC was unable to find many views that were exact syntactic matches, whereas BFR was able to exploit additional views due to BFR’s ability to reuse and re-purpose previous results through semantic query rewriting. Although this result is workload dependent, this example highlights the fact that while rewriting identical results is clearly beneficial, our approach completely subsumes those that only reuse syntactically identical results: even when there are no identical views our method may still find a low-cost rewrite. To further illustrate this, we next perform an additional experiment after removing all identical views from the system before applying our BFR algorithm.

Table 2: Execution time improvement without identical views.

<table>
<thead>
<tr>
<th></th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
<th>$A_4$</th>
<th>$A_5$</th>
<th>$A_6$</th>
<th>$A_7$</th>
<th>$A_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFR</td>
<td>57%</td>
<td>64%</td>
<td>83%</td>
<td>85%</td>
<td>51%</td>
<td>96%</td>
<td>88%</td>
<td>84%</td>
</tr>
<tr>
<td>BFR-SYNTACTIC</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Here we repeat the user evolution experiment after discarding from the system all views that are identical to a target in each of the hold-out queries ($A_{1v4}$). Without these views, syntactic caching-based methods will not be able to find any rewrites, resulting in 0% improvement. Table 2 reports the percentage improvement for each analyst $A_1$–$A_8$ after discarding all identical views. This shows BFR continues to reduce query execution time dramatically even when there are no views in the system that are an exact match to the new queries. The performance improvements are comparable to the result in Figure 8(c) which represents the same experiment without discarding the identical views. Notably there is a drop for $A_2$ compared to the results reported in Figure 8(c) for $A_1$. This is because previously in Figure 8(c), $A_2$ had benefited from an identical view corresponding to a restaurant-similarity computation that it now has to recompute. The identical views discarded constituted only 7% of the view storage space in this experiment, indicating there are many other useful views.
Given that analysts pose different but related queries, any method that relies solely on identical matching may have limited benefit.

9. RELATED WORK

Query Rewriting Using Views. There is a rich body of previous work on rewriting queries using views, but these only consider a restricted class of queries. Representative work includes the popular algorithm MiniCon [23], recent work [15] showing how rewriting can be scaled to a large number of views, and rewriting methods implemented in commercial databases [7, 29]. However, in these works both the queries and views are restricted to the class of conjunctive queries (SPJ) or additionally include groupby and aggregation (SPJG).

Our work differs in the following two ways: (a) We show how UDFs can be included in the rewrite process using our UDF model, which results in a unique variant of the rewrite problem when there is a divergence between the expressivity of the queries and that of the rewrite process; (b) Our rewrite search process is cost-based—OprYCost enables the enumeration of candidate views based on their ability to produce a low-cost rewrite. In contrast, traditional approaches (e.g., [7, 23]) typically determine containment first (i.e., if a view can answer a query) and then apply cost-based pruning in a heuristic way. This unique combination of features has not been addressed in the literature for the rewrite problem.

Online Physical Design Tuning. Methods such as [25] adapt the physical configuration to benefit a dynamically changing workload by actively creating or dropping indexes/views. Our work is opportunistic, and simply relies on the by-products of query execution that are almost free. However, view selection methods could be applicable during storage reclamation to retain only those views that provide maximum benefit.

Reusing Computations in MapReduce. Other methods for optimizing MapReduce jobs have been introduced such as those that support incremental computations [20], sharing computation or scans [21], and re-using previous results [6]. As shown in Section 8.3.4, our approach completely subsumes these methods.

Multi-query optimization (MQU). The goal of MQO [26] (and similar approaches [21]) is to maximize resource sharing, in particular common intermediate data, by producing a scheduling strategy for a set of in-flight queries. Our work produces a low-cost rewrite rather than a schedule for concurrent query plans.

10. DISCUSSION AND CONCLUSION

Big data analysis frequently includes exploratory queries that contain UDFs. Hence, to exploit previous results, a semantic understanding of UDF computation is required. In this work, we presented a gray-box UDF model that is simple but expressive enough to capture a large class of big data UDFs, enabling our system to effectively exploit prior computation. We also presented a rewrite algorithm that efficiently explores the large space of views.

Retaining opportunistic views within a limited storage space budget requires navigating the tradeoff between storage cost and query performance, which is equivalent to the view selection problem. During our experiments, accumulating all views for every query resulted in an additional storage space of only 2.0× the base data size (≈2TB). The relatively small total size of all the views with respect to the log base data is due to several reasons. First, the logs are very wide, as they record a large number of attributes. However, a typical query only consumes a small fraction of these log attributes, which is consistent with observations in big data systems. Second, it is not uncommon for the log attributes to have missing values, since the data may be dirty or incomplete. For instance, in the Twitter log, a tweet may have missing location values, which a query may discard.

Developing a good view selection policy in this space is an interesting area of future work. One could consider access-based policies such as LRU and LFU, or cost-based methods often used for physical design tuning. In the extended version [17] we show that the rewriter performs well even with a trivial storage reclamation policy, while in [18] we address a variant of this problem that considers several policies.

Acknowledgement: We thank Jeffrey Naughton for his insightful comments on the earlier versions of the paper. We also thank the anonymous SIGMOD reviewers for their useful feedback. J. LeFevre and N. Polyzotis were partially supported by NSF grant IIS-0447966 and a research grant from NEC Laboratories America.

11. REFERENCES