Optimization for iterative queries on MapReduce

Makoto Onizuka*, Hiroyuki Kato†, Soichiro Hidaka†, Keisuke Nakano‡, Zhenjiang Hu†
*NTT Software Innovation Center, †National Institute of Informatics, ‡University of Electro-Communications
onizuka.makoto@lab.ntt.co.jp, {kato, hidaka}@nii.ac.jp, ksk@cs.uec.ac.jp, hu@nii.ac.jp

ABSTRACT

We propose OptIQ, a query optimization approach for iterative queries in distributed environment. OptIQ removes redundant computations among different iterations by extending the traditional techniques of view materialization and incremental view evaluation. First, OptIQ decomposes iterative queries into invariant and variant views, and materializes the former view. Redundant computations are removed by reusing the materialized view among iterations. Second, OptIQ incrementally evaluates the variant view, so that redundant computations are removed by skipping the evaluation on converged tuples in the variant view. We verify the effectiveness of OptIQ through the queries of PageRank and k-means clustering on real datasets. The results show that OptIQ achieves high efficiency, up to five times faster than is possible without removing the redundant computations among iterations.

1. INTRODUCTION

Many Web-based companies such as Google, Facebook, Yahoo are intensively analyzing tremendous amounts of data, such as user contents and access logs. MapReduce [4], which is a distributed computation framework, is widely used for statistical analysis. MapReduce works on hundreds or thousands of commodity machines and assures high scalability and availability. For example, dozens to hundreds of TB datasets are processed daily by MapReduce at Google [5], and 75TB of compressed data is processed every day by Hadoop at Facebook [27].

In detail, MapReduce allows map functions to be applied to the data stored in a distributed file system, resulting in key-value pairs. All the produced pairs are routed (shuffled) to reduce tasks according to the output key, so that all pairs with the same key wind up at the same reduce task. The reduce tasks apply reduce functions to the values associated with one key and produce a single result for that key. Thus, the programmers can implement various types of distributed algorithms on MapReduce by implementing the map and reduce functions; MapReduce isolates the programmers from the difficulties caused by the distributed processing; how data and processes are distributed to the servers and how to handle failures and straggles of servers.

Proceedings of the VLDB Endowment, Vol. 7, No. 4


```
1: class Mapper
2:   method Map(nid n; node N)
3:     p ← N.PageRank/|N.AdjacencyList|
4:     Emit(nid n, N) // Pass along graph structure
5:     for all nodeid m ∈ N.AdjacencyList do
6:       Emit(nid m, p) // Pass PageRank mass to neighbors

1:  class Reducer
2:    method Reduce(nid m, [p1,p2,...])
3:      M ← φ
4:      for all p ∈ counts [p1,p2,...] do
5:        if IsNode(p) then
6:          M ← p // Recover graph structure
7:        else
8:          s ← s + p // Sum incoming PageRank contributions
9:      M.PageRank ← s
10:     Emit(nid m, node M)
```

Figure 1: PageRank program (single iteration) in MapReduce

The active application fields of the MapReduce framework include machine learning algorithms [24, 26, 25, 12, 21]. Shim presented in his tutorial talk in [24] the state-of-the-art in MapReduce algorithms for data mining, machine learning, and similarity joins. Chu et al. [25] describe a method of rewriting various types of machine learning algorithms into summation form, which can be directly transformed to the MapReduce framework. Mahout [26] provides libraries of machine learning algorithms. In particular, there are many iterative algorithms for machine learning, which are common in clustering and graph mining; PageRank, random walk with restarts (RWR), k-means clustering, non-negative matrix factorization (NMF) [14].

However, it is still difficult for programmers to implement iterative algorithms efficiently on the MapReduce framework, because improving their response time is not a trivial task. For example, consider the PageRank computation on MapReduce [16]; PageRank is computed by iteratively executing the program in Figure 1 [1] until convergence. The map function evenly divides each node's PageRank score, `N.PageRank`, and passes each fraction to neighbor `m` along outgoing edges. The reduce function sums up the divided scores, `p1`, `p2`, ..., at each destination node `m`. Which parts of this program are inefficient? There are two types of redundant computations. First, the map function shuffles the graph structure (line 4 in map function) in every iteration. This is redundant and the redundancy is removed by reusing the shuffled result of the graph structure. Second, the PageRank scores of the nodes are computed in every iteration, even if many of them have already converged in earlier iterations. This is also redundant and can be improved by computing PageRank incrementally, that is to skip computing the PageRank scores of converged nodes. However, it is burden

[1]This program is a simplified version of PageRank since the authors of [16] ignore the random jump factor and dangling nodes.

for programmers to manually program applications by identifying and removing the above redundant computations. Several techniques have been proposed for efficient query processing of iterative queries on distributed frameworks, Spark [28], HaLoop [3], REX [19]. None of them are automatic and general framework for query optimization. Those former approaches force programmers to identify and remove redundant computations manually by integrating caching or incremental techniques into programs. In detail, Spark and HaLoop cache and reuse input files to map functions and reduce functions across iterations, however the programmers have to manually design invariant tables and specify those tables to be cached during iterations. REX achieves incremental computation, however the programmers have to manually identify which parts of the programs are incrementally computable and indicate how those parts should be incrementally computed. For the above problem, the traditional query optimization techniques do not identify and remove the redundant computations among different iterations.

We open up a new research field of optimization for iterative queries. We tackle this field by integrating the traditional optimization techniques in database and compiler areas. We extend SQL language to express iterative queries and propose a query optimization approach, OptIQ ([Opt]imization for [I]terative [Q]ueries), for iterative queries with convergence property. The novelty of OptIQ is that it provides general framework for removing redundant computations identified in iterative queries. Since SQL is a declarative high level language that hides low level execution plans, the programmers are freed from the burden of manually identifying and removing redundant computations in iterative queries. OptIQ automatically removes the redundant computations in iterative queries by extending the traditional techniques of view materialization and incremental view evaluation in database area and those of program analysis and transformation in compiler area. First, OptIQ decomposes iterative queries into invariant and variant views, and materializes the former view for reusing it in the following iterations. Second, OptIQ incrementalizes the variant views by skipping the evaluation of converged tuples. OptIQ is fully automatic and minimizes redundant computations efficiently.

Our contributions are summarized as follows:

- We identify a problem of redundant computations in processing iterative queries, which is a typical problem in iterative algorithms, such as clustering and graph mining.
- We propose OptIQ, an automatic query optimization technique for iterative queries with convergence property. For a given iterative query, OptIQ removes redundant computations among different iterations by extending the traditional techniques of view materialization and incremental view evaluation in database area and those of program analysis and transformation in compiler area.
- We implement OptIQ on MapReduce and Spark. We also report experiments on iterative queries of PageRank and k-means clustering over real datasets. The results show that OptIQ achieves high efficiency both on MapReduce and Spark, up to five times faster than is possible without removing the redundant computations among iterations.

The rest of this paper is organized as follows. Section 2 defines a query language for iterative queries. OptIQ is introduced in Section 3. Section 4 describes how queries are compiled into MapReduce programs. Section 5 reports the results of experiments on clustering and graph mining. Section 6 addresses related work and Section 7 concludes this paper.

initialize {*initialize temporal tables*}∗
iterate
{{**set**|**let**} *update table* = *step query*}∗
until *condition* **on** *update table*
return {*result construction*}∗

Figure 2: Iterative query syntax

2. ITERATIVE QUERY LANGUAGE

We define a query language for iterative queries. Figure 2 shows the syntax, which is influenced by MRQL [7].

A query consists of three parts; The **initialize** clause initializes tables, which are used in step queries. The **iterate** clause specifies step queries and update tables; **set** and **let** statements specify that the result of step query is set to *global table* and *local table*, respectively. While the local table stores the tuples computed in the current iteration, the global table stores the tuples computed in the current and previous iterations. We refer to tuples in the current iteration as new tuples and those in the previous iteration as old tuples by using prefixes `new` and `old`, respectively. The difference between global table and local table affects query optimization, particularly in distributed environments. Global tables have to be stored in a distributed file system so that they are accessed from different iterations, whereas local tables can be stored in a local file system for optimization. We assume that the programmers use **set** statement only and that it may be replaced by the **let** statement or vice versa during query optimization. The step queries are iteratively evaluated until convergence. The **until** clause specifies the convergence condition on the global table. A convergence condition is expressed by comparing old and new tuples of an update table and it is tested against all the tuples of the update table. The old and new tuples are related by the key as follows; tuple $\mathtt{r}_1$ accessed by `new` is the new version of tuple $\mathtt{r}_2$ accessed by `old`, if they share the same key. The **return** clause constructs a query result after the convergence.

Step queries are expressed by SQL statements whose operators are given as follows:

$$\begin{aligned} projection &: T(\mathtt{a}_1, ..., \mathtt{a}_n) = \pi_{\mathtt{a}_1,...,\mathtt{a}_n}(\mathtt{R}) \\ selection &: T(\mathtt{schema}(\mathtt{R})) = \sigma_{\phi}(\mathtt{R}) \\ join &: T(\mathtt{schema}(\mathtt{R}) \cup \mathtt{schema}(\mathtt{S})) = \mathtt{R} \bowtie_{\phi} \mathtt{S} \\ group-by &: T(\mathtt{a}_1, ..., \mathtt{a}_n, \mathtt{m}) = {}_{\mathtt{a}_1,...,\mathtt{a}_n}\mathcal{G}_{\mathtt{m}=\mathtt{f}(\tau)}(\mathtt{R}) \end{aligned}$$

where R and S are input tables, $T(\mathtt{list})$ is a table with `list` attributes, $\mathtt{schema}(\mathtt{R})$ are the attributes of R, $\mathtt{a}_1, ..., \mathtt{a}_n \subseteq \mathtt{schema}(\mathtt{R})$, $\tau \subseteq \mathtt{schema}(\mathtt{R})$, and ϕ is a propositional formula.

The projection operation projects a set of specified attributes, $\mathtt{a}_1$, ..., $\mathtt{a}_n$, in the input table. The selection operation extracts the tuples that satisfy propositional formula ϕ in the input table. The join operation computes the cross product of two input tables and extracts the tuples that satisfy ϕ[2]. The resulting schema is the union of the schema of the input tables. The group-by operation constructs groups of tuples and computes aggregate functions, so that the tuples within the same group share the same values of key attributes, $\mathtt{a}_1, ..., \mathtt{a}_n$, and those tuples are put into aggregation function $\mathtt{f}(\tau)$ and the result is set to attribute `m`. The resulting schema consists of $\mathtt{a}_1, ..., \mathtt{a}_n$ and `m`.

EXAMPLE 2.1 (PAGERANK). We consider a query for the simplified version of PageRank in Figure 1 without any loss in

[2]We use $\bowtie$ as a natural join and $R \bowtie_a S$ as a simplified form of $R \bowtie_{R.a=S.a} S$.

generality. We apply OptIQ to the original PageRank and conduct experiments in Section 5.

Schema:

```
1: Graph(src,dest,score)
2: Count(src,count)
3: Score(dest,score)
```

Query:

```
1: initialize
2:  setup Graph';
3:  Count = select src,count(dest) as count
4:          from Graph
5:          group by src;
6: iterate
7:  set Score =
8:    select n.dest,sum(n.score/Count.count) as score
9:    from Graph' as n, Count
10:   where n.src = Count.src
11:   group by n.dest;
12: set Graph' =
13:   select m.src,m.dest,Score.score
14:   from Graph' as m, Score
15:   where m.src = Score.dest;
16: until |new.score - old.score| < ε on Score;
17: return Score;
```

Graph table has attributes of the identity of node src, the destination node dest of src, and the PageRank score score of src. Count and Score are tables derived from Graph. src and dest are keys in Count and Score, respectively. PageRank for Graph is computed as follows. The initialize clause copies Graph′ from Graph and the score of the nodes are initialized as the inverse of the number of nodes by following the definition of PageRank. Count is defined by the query in lines 3-5, which computes the number of out-going edges for each node src. There are two step queries in the iterate clause starting from line 7 with a convergence condition in line 16. The iteration is terminated when the difference between the new scores and the old scores of Score are smaller than threshold ε. The first query (lines 8-11) computes a new PageRank score for each destination node n.dest by summing up the divided scores n.score/Count.conunt of incoming source nodes. The result is set to Score. The second query (lines 13-15) updates the old scores of Graph′ by the new scores of Score by joining Score and Graph′. □

EXAMPLE 2.2 (k-MEANS CLUSTERING). A query for computing k-means clustering is given as follows.

Schema:

```
1: Centroid(id,pos)
2: Point(id,cid,pos)
```

Query:

```
1: initialize setup Centroid; Point' = Point;
2: iterate
3:  set Point' =
4:    select p.id,
5:          (select c.id
6:           from Centroid as c
7:           order by distance(c.pos,p.pos)
8:           limit 1) as cid,p.pos
9:    from Point' as p;
10:  set Centroid =
11:   select cid as id, avg(pos) as pos
12:   from Point'
13:   group by cid;
14: until |new.pos - old.pos| < ε on Centroid;
15: return Centroid;
```

There are two tables, Centroid and Point. Point is a collection of data points to be clustered. Centroid is a collection of the centroids of the clusters. Centroid has the attributes of identity id and position pos. Point has the attributes of identity id, identity of Centroid to which Point belongs to, and position pos. The initialize clause initializes Centroid as k points randomly chosen from Point (line 1). The iterate clause starts at line 2 and contains a convergence condition (line 14); the iteration is terminated when the difference between new and old positions of Centroid is smaller than ε. There are two step queries in lines 3-13. The first query locates the closest centroid in Centroid to every data point in Point. The second query groups data points in Point by the closest centroid, computes the average position of the data points in each group, and sets it as a new position of the centroid. □

3. QUERY OPTIMIZATION

OptIQ is a query optimization technique for iterative queries; it removes redundant computations among different iterations. Here we have a question: "What are redundant computations among iterations?" The redundant computations are the operations on unmodified attributes of tuples or on attributes of unmodified tuples among iterations. OptIQ enhances two traditional optimization techniques for SQL queries: view materialization and incremental view evaluation. Both techniques reuse the results of step queries. The view materialization technique reuses the result of subqueries for unmodified attributes. The incremental view evaluation technique reuses the result of queries for unmodified tuples. The ideas that underlie the removal of redundant computations are as follows.

Table decomposition & view materialization detects modified attributes among iterations, extracts maximum subqueries in iterative queries that do not access any modified attributes (we call the subqueries invariant views), and materializes the subqueries. The input queries are rewritten to use the invariant views for efficient query processing.

Automatic incrementalization detects modified tuples (delta table) among iterations, derives incremental queries for input queries, and thus incrementally evaluates them for the modified tuples. This incrementalization optionally utilizes filtering function to reduce the number of the modified tuples according to convergence condition. Actually, various researches [19, 17] employ this idea. We leave users to choose to use this filtering function, since there is a trade-off between speed and accuracy.

Figure 3 overviews OptIQ. The left part shows the original query execution flow of an iterative query. The middle part shows a query execution flow after applying table decomposition & view materialization. Invariant views are extracted from the original query and they are constructed before the iterations. Variant views are repeatedly evaluated until convergence. The right part shows a query execution flow based on the incremental evaluation; a variant view is incrementally evaluated (indicated by "+=" and "delta" in the figure) during iterations.

3.1 View materialization

OptIQ optimizes iterative queries by employing view materialization in the following steps: 1) decompose input update tables

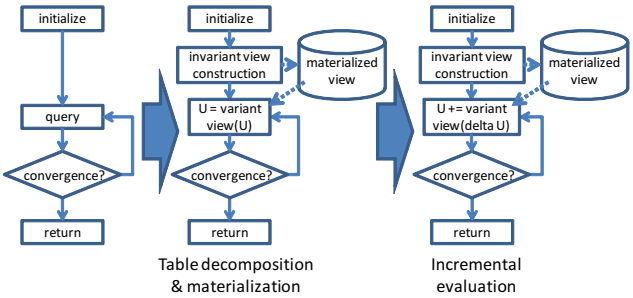


Figure 3: Overview of OptIQ

into variant and invariant tables, and rewrite the iterative query such that the update tables are replaced by variant tables. 2) materialize partial operations in the iterative query (we call it invariant view) that accesses invariant tables, and rewrite and simplify the iterative query in which the invariant view is reused. In this subsection, we will explain the above two-step view materialization by using the PageRank iterative query in Example 2.1 as a running example. We show how to rewrite a k-means iterative query after that.

3.1.1 Table decomposition

An iterative query is executed by updating (possibly large) tables, called *update tables*, at each step. The purpose of table decomposition is to remove redundant computations by splitting the update table into a variant and an invariant one. A variant table contains 'modified' attributes of the original table which may be updated during a single step computation. An invariant table contains not only 'unmodified' attributes of the original table which are not updated during computation but also new attributes which store values derived from the unmodified attributes. Additionally, both tables share an unmodified attribute such that the original table can be obtained by joining variant and invariant tables using the attribute.

Modified and unmodified attributes are discriminated by a conservative analysis: we judge an attribute to be unmodified only when the attribute is projected at the **set** statement of the update table. In the running example, the `Graph'` table is an update table that has three attributes, `score`, `src`, and `dest`. Since `src` and `dest` are just projected at the **set** statement of `Graph'` (lines 12-15), they are unmodified attributes while the `score` attribute is modified. Even though more attributes might be found unmodified through further analysis, we take a conservative approach to this discrimination step.

Next we decompose the update table according to the discrimination of modified and unmodified attributes. We have two update tables, `Score` and `Graph'`. Since `Score` has no unmodified attribute except its key `dest`, we do not decompose the `Score` table. Let us decompose the `Graph'` table. We choose an unmodified attribute of the update table. The attribute is used for synthesizing the update table from variant and invariant tables by joining them. For the running example, by letting `src` be the key attribute, we have two tables `VT(src,score)` and `IT(src,dest)` such that `Graph'` $=$ `VT` $\bowtie_{\texttt{src}}$ `IT`.

We rewrite the iterative query by replacing the `Graph'` table with `VT` $\bowtie_{\texttt{src}}$ `IT`. First we initialize variant and invariant tables instead of the update table in the **initialize** clause. Next, we rewrite all subqueries by flattening them in the following ways:

- We rewrite the attributes of the `VT` $\bowtie_{\texttt{src}}$ `IT` table into those of either `VT` or `IT`. Regarding the `src` attribute, we may rewrite it into either `VT.src` or `IT.src` according to whether the next step can yield further improvement.

Schema:

```
Graph(src,dest,score)
Count(src,count)
Score(dest,score)
VT(src,score)
IT(src,dest)
```

Query:

```
 1: initialize VT = select src,score from Graph;
 2:          IT = select src,dest from Graph;
 3:          Count = select src,count(dest)
 4:                  from IT
 5:                  group by src;
 6: iterate
 7:   set Score =
 8:     select IT.dest,sum(VT.score/Count.count) as score
 9:     from VT, IT, Count
10:     where IT.src = Count.src
11:           VT.src = IT.src
12:     group by IT.dest;
13:   set VT =
14:     select VT.src,Score.score
15:     from VT, Score
16:     where VT.src = Score.dest;
17: until |new.score - old.score| < ε on Score;
18: return Score;
```

Figure 4: PageRank query after table decomposition

- We replace **from** `VT` $\bowtie_{\texttt{src}}$ `IT` table with **from** `VT`, `IT` and add `VT.src = IT.src` to the **where** clause. If none of the attributes of the `VT` (resp. `IT`) table are referred to, we just replace the **from** clause with **from** `IT` (resp. **from** `VT`).

For example, the subquery

```
select n.dest,sum(n.score/Count.count) as score
from (VT ⋈src IT) as n, Count
where n.src = Count.src
group by n.dest
```

is rewritten into

```
select IT.dest,sum(VT.score/Count.count) as score
from VT, IT, Count
where IT.src = Count.src
      VT.src = IT.src
group by IT.dest
```

This yields the query in Figure 4.

3.1.2 Subquery lifting

So far we have decomposed the attributes into unmodified and modified ones, and decomposed the table into invariant and variant tables. The step queries were rewritten in terms of these tables.

The next step is to construct read-only materialized views that are accessed by the step queries. We call them *invariant views*. Since an invariant view is constructed only once across the entire step queries, it is constructed in the **initialize** clause in the iterative query. The most primitive way of constructing an invariant view is just to project the unmodified attributes, and this is already achieved at the time of the table decomposition (Figure 4). Moreover, we could remove more redundancy by extracting loop-invariant computations using unmodified attributes in the iteration. This optimization is called *loop invariant code motion* [2] in the traditional compiler literature, and we achieve in OptIQ by the following three steps.

Step 1: Constant let statement lifting. Lift **let** statements in the **iterate** clause to the **initialize** clause, if they are *constant* queries, i.e., all of their computations depend only on the invariant tables (we do not have such a lifting opportunity in the running example).

Schema:

```
Graph(src,dest,score)
Count(src,count)
Score(dest,score)
IT(src,dest)
IT_Count(src,dest,count)
```

Query:

```
 1: initialize IT = select src,dest from Graph;
 2:      Count = select src,count(dest)
 3:              from IT group by src;
 4:   IT_Count = select IT.src,IT.dest,Count.count
 5:              from IT, Count
 6:              where IT.src = Count.src;
 7: iterate
 8:   set Score =
 9:     select ic.dest,sum(sc.score/ic.count) as score
10:     from Score as sc, IT_Count as ic
11:     where sc.dest = ic.src
12:     group by ic.dest;
13: until |new.score - old.score| < ε on Score;
14: return Score;
```

Figure 5: PageRank query after subquery lifting

Step 2: Invariant subquery lifting. In addition, we can extract and materialize invariant subqueries used in the computation of tables including modified attributes in the **set** and **let** statements. The enclosing queries are rewritten accordingly using the materialized views.

In the running example, since the lifted table `Count` is always used for joining with the invariant table `IT` at the **set** statement for `Score` (lines 7-12), we can preliminarily join the `IT` and `Count` tables to form (in addition to `IT`) another materialized view `IT_Count` as follows:

```
IT_Count = select IT.src,IT.dest,Count.count
           from IT, Count
           where IT.src = Count.src.
```

Correspondingly, the **set** statement is rewritten by replacing the tables `IT` and `Count` in the **from** and **select** clauses by `IT_Count`, and removing the join condition in the **where** clause corresponding to the above join, to yield

```
set Score =
  select ic.dest,sum(VT.score/ic.count) as score
  from VT, IT_Count as ic
  where VT.src = ic.src
  group by ic.dest.
```

Step 3: Common subquery elimination. If we have common subqueries in the **initialize** clause, we can further factor them out to reduce the redundancy within the clause (we do not have such an elimination opportunity in the running example). We also apply other common query rewriting techniques such as unnesting or identity query elimination, whenever possible including the previous and the following steps.

In the running example, the **set** statement computing `VT` (lines 13-16) can be rewritten into

```
set VT = Score
```

assuming that the set of `src` in `VT` and the set of `dest` in `Score` are identical. Correspondingly, we can replace `VT`s with `Score` in the **set** statement updating `Score` (`VT.src` should be replaced by `Score.dest`), and remove the definitions of `VT` from the **initialize** clause and **set** statement entirely.

After these three steps, we obtain the query in Figure 5. Note that we may apply further grouping for the materialized views for efficient MapReduce computation. We revisit this optimization in Section 4.2.

Schema:

```
Centroid(id,pos)
Point(id,cid,pos)
VT(id,cid)
IT(id,pos)
```

Query:

```
 1: initialize
 2:   IT = select id,pos from Point
 3: iterate
 4:   let VT =
 5:     select IT.id,
 6:            (select c.id
 7:             from Centroid as c
 8:             order by distance(c.pos,IT.pos)
 9:             limit 1) as cid
10:     from IT
11:   set Centroid =
12:     select VT.cid as id, avg(IT.pos)
13:     from VT, IT
14:     group by VT.cid
15:     where VT.id = IT.id
16: until |new.pos - old.pos| < ε on Centroid;
17: return Centroid;
```

Figure 6: k-means query after rewriting

3.1.3 *Other example: k-means query*

We apply our query rewriting to another example, k-means clustering, see Example 2.2.

In this query, two tables, `Point'` and `Centroid`, are the update tables. The `Point'` table has two unmodified attributes, `id` and `pos`, while the `Centroid` table has none. Hence, we only apply table decomposition to the `Point'` table.

We choose the unmodified attribute `id` for table decomposition because it is a primary key. We rewrite the `Point'` table into $\texttt{VT} \bowtie_{\texttt{id}} \texttt{IT}$ by using the variant table `VT(id,cid)` and the invariant table `IT(id,pos)`.

The **set** statement for `Point'` is rewritten into the **set** statement for `VT` as follows:

```
set VT =
  select IT.id,
         (select c.id
          from Centroid as c
          order by distance(c.pos,IT.pos)
          limit 1) as cid
  from IT;
```

Although the **select** `IT.id` clause could be **select** `VT.id`, we use `IT.id` so that the join operation (**from** `VT.id = IT.id`) is not required.

The rewritten query is shown in Figure 6, where we apply further simplification as follows. Note that in the iteration body, the `VT` table computed at the previous iteration is never referred to. Hence, we do not have to initialize the `VT` table and we can use just **let** rather than **set** for `VT`.

3.2 Automatic incrementalization

In this section, we propose a specific automatic incrementalization technique for optimization of iterative queries; it automatically detects the modified tuples (delta tables) among iterations and incrementally evaluates the iterative query for those tuples.

3.2.1 Update operations

Before explaining how to evaluate iterative queries incrementally, we should elucidate what kind of changes we consider (allow) on the tables (relational databases). There are basically three kinds of changes on tables, tuple insertion, tuple deletion, and tuple update. A *tuple insertion*

$$T \uplus \Delta$$

adds a disjoint set of tuples in Δ to table T, a *tuple deletion*

$$T \setminus \Delta$$

deletes a set of tuples in Δ from table T, and a *tuple update*

$$T \oplus_m \Delta$$

updates tuples of T by replacing the value of attribute m by "adding" it to the corresponding m value in Δ.

It is worth noting that tuple updates appear more often than the other two changes in the context of iterative queries, in which a table is iteratively updated until it reaches a stable form. We can define tuple update generally as follows:

$$T \ \oplus_{m_1,\ldots,m_k} \ \Delta \stackrel{\text{def}}{=} \pi_{(T.m_1+\Delta.m_1),\ldots,(T.m_k+\Delta.m_k),u_1,\ldots,u_{k'}}(T \bowtie \Delta)$$

where m_i and u_j denote modified attributes and unmodified attributes in the update table T, respectively. For instance, given a personal table we may express raising a worker's salary and his position level by $Personal \oplus_{salary,level} Up$. Note that we sometimes omit modified attributes on $\oplus$ when they are not used. Dually, we can define the following:

$$T_1 \ominus_{m_1,\ldots,m_k} T_2 \stackrel{\text{def}}{=} \pi_{(T_1.m_1-T_2.m_1),\ldots,(T_1.m_k-T_2.m_k),u_1,\ldots,u_{k'}}(T_1 \bowtie T_2)$$

where T_1 and T_2 have a common schema.

3.2.2 Detecting delta tables

It turns out to be rather straightforward to detect delta tables after our table decomposition as discussed in Section 3.1, where a set of modified attributes have been identified. For variant table T with modified attributes $m_1, \ldots, m_n$, we define a tuple update on T by

$$T \ \oplus_{m_1,\ldots,m_n} \ \Delta_T$$

where overloaded $+$ (used in the definition of $\oplus$ as given before) on the values of attribute m_i is defined as follows:

- If m_i is a numeric attribute, $+$ is the common addition operation.
- If m_i is an enumeration attribute, $+$ is defined as replacing the original value by a new one (i.e., $a + b = b$).

3.2.3 Deriving incremental queries

A lot of work is being devoted to deriving incremental queries for the insertion and deletion of tuples in relational databases by using the notion of delta tuples [9] or ring structure [13]. In iterative computation, however, updates (changes of values of tuples) in the iteration are more frequent than deletion or insertion [19]. This leads us to look into new ways to deal with value updates for deriving incremental queries.

Consider the following simple but general **iterate** clause:

$$\begin{array}{l}\textbf{iterate}\\ \quad \texttt{set}\ T = \mathtt{q}(T)\\ \textbf{until}\ \phi(\Delta_T)\end{array} \tag{1}$$

where T is an update table, and $\mathtt{q}(T)$ is a query (written in relational algebra) to table T, and $\phi(\Delta_T)$ denotes a termination condition on delta table Δ_T.

We would like to see how to efficiently compute

$$\texttt{set}\ T = \mathtt{q}(T \oplus \Delta_T)$$

iteratively. To this end, we distribute the computation of q into $T \oplus \Delta_T$ and investigate new ways of incrementalizing aggregation computations in group-by. These form the core of our automatic incrementalization algorithm.

Query distribution over tuple updates

Suppose that q is distributed over $\oplus$ by $\otimes$:

$$q(T \oplus \Delta_T) = q(T) \otimes q(\Delta_T).$$

Then, we can rewrite the iterate-clause (1) into the following iterative query:

$$\begin{array}{l}\textbf{initialize}\\ \quad \Delta_T = \mathtt{q}(T) \ominus T\\ \textbf{iterate}\\ \quad \texttt{set}\ T = T \otimes \Delta_T\\ \quad \texttt{set}\ \Delta_T = \mathtt{q}(\Delta_T)\\ \quad \texttt{set}\ \Delta_T = \psi(\Delta_T, C)\\ \textbf{until}\ \phi(\Delta_T)\end{array} \tag{2}$$

if Δ_T obtained in the initialization does not satisfy ϕ. As a matter of fact, even if the initial Δ_T satisfies ϕ (which seldom happens in practice), it is still fine to use (2) because computing Δ_T once again dose not matter when Δ_T converges. Note that ψ is an optional filtering function for adapting Δ_T according to context C. We have to pay attention that introducing ψ may change the accuracy of the result. In our PageRank example as will be shown in Example 3.1, the termination condition ϕ is utilized as the filtering function ψ for adapting Δ_T to reduce the size of Δ_T; the result of the incremental version is slightly different from the original one. We will discuss this in Section 5.3.1. Also, Adaptive PageRank [11] can be emulated by using ψ with an iterative parameter in a suitable way.

In general, q may not be fully distributed over $\oplus$, but if we could distribute it to some extent and extract some computations on T out, we could reuse the computation results on T in the iteration steps. Figure 7 summarizes the rules for this distribution. We use $\bar{a}$ to denote a set of attributes. When an input query is not distributed over $\oplus$, we abandon incrementalizing it. For the selection operator, the distributive law holds if there are no common attributes between the attributes used in the selection conditions and the modified attributes. The rules for projection and join are easy to understand.

For the group-by operator, it can be distributed over $\oplus$ if the aggregation function is `sum` over the modified attribute b. We omit rules for other aggregation functions such as `count`.

EXAMPLE 3.1 (PAGERANK INCREMENTALIZATION). Consider incrementalizing the PageRank query shown in Figure 5. The query body (lines 9 - 12) in the iterate clause can, in terms of relational algebra with incorporation of the delta table $\Delta_{\texttt{Score}}$, be rewritten as follows by using the distributive laws listed in Figure 7 where `IC` and `SC` denote `IT_Count` and `Score`, respectively.

$$\frac{\bar{a} \cap \bar{b} = \emptyset}{\sigma_{p(\bar{a})}(T \ \oplus_{\bar{b}} \ \Delta_T) = \sigma_{p(\bar{a})}(T) \ \oplus_{\bar{b}} \ \sigma_{p(\bar{a})}(\Delta_T)} \ \text{SEL}$$

$$\frac{\bar{a} \supseteq \bar{b}}{\pi_{\bar{a}}(T \ \oplus_{\bar{b}} \ \Delta_T) = \pi_{\bar{a}}(T) \ \oplus_{\bar{b}} \ \pi_{\bar{a}}(\Delta_(T))} \ \text{PROJ1}$$

$$\frac{\bar{a} \cap \bar{b} = \emptyset}{\pi_{\bar{a}}(T \ \oplus_{\bar{b}} \ \Delta_T) = \pi_{\bar{a}}(T)} \ \text{PROJ2}$$

$$\frac{\bar{a} \cap \bar{b} = \emptyset \quad \bar{c} \cap \bar{b} = \emptyset}{\begin{array}{l}(T_1 \ \oplus_{\bar{a}} \ \Delta_{T_1}) \bowtie_{p(\bar{b})} (T_2 \ \oplus_{\bar{c}} \ \Delta_{T_2}) \\ \quad = (T_1 \bowtie_{p(\bar{b})} T_2) \oplus_{\bar{a}} \ (\Delta_{T_1} \bowtie_{p(\bar{b})} T_2) \\ \qquad \oplus_{\bar{c}} \ (T_1 \bowtie_{p(\bar{b})} \Delta_{T_2})\end{array}} \ \text{JOIN}$$

$$\frac{}{\begin{array}{l}{}_{\bar{a}}\mathcal{G}_{m=\texttt{sum}(b)}\,(T \ \oplus_b \ \Delta_T) \\ \quad = \ ({}_{\bar{a}}\mathcal{G}_{m=\texttt{sum}(b)}\,(T)) \\ \qquad \oplus_m \ ({}_{\bar{a}}\mathcal{G}_{m=\texttt{sum}(b)}\,(\Delta_T))\end{array}} \ \text{GROUPBY}$$

Figure 7: Distribution rules for incrementalization

Also, the join predicate ϕ denotes `Score.dest` $=$ `IT_Count.src`, and `sum(T)` denotes `sum(score/count)`.

$$\begin{array}{l}\pi_{\texttt{IC.dest, SC.score}} \\ \quad ({}_{\texttt{IC.dest}}\mathcal{G}_{\texttt{score=sum(T)}}((\texttt{SC} \ \oplus_{\texttt{score}} \ \Delta_{\texttt{SC}}) \bowtie_\phi \texttt{IC})) \\ = \{\text{by JOIN}\} \\ \pi_{\texttt{IC.dest, SC.score}} \\ \quad ({}_{\texttt{IC.dest}}\mathcal{G}_{\texttt{score=sum(T)}}((\texttt{SC} \bowtie_\phi \texttt{IC}) \oplus_{\texttt{score}} (\Delta_{\texttt{SC}} \bowtie_\phi \texttt{IC}))) \\ = \{\text{by GROUPBY}\} \\ \pi_{\texttt{IC.dest, SC.score}} \\ \quad (({}_{\texttt{IC.dest}}\mathcal{G}_{\texttt{score=sum(T)}}(\texttt{SC} \bowtie_\phi \texttt{IC})) \\ \quad \oplus_{\texttt{score}} \\ \quad ({}_{\texttt{IC.dest}}\mathcal{G}_{\texttt{score=sum(T)}}(\Delta_{\texttt{SC}} \bowtie_\phi \texttt{IC}))) \\ = \{\text{by PROJ1}\} \\ \pi_{\texttt{IC.dest, SC.score}}({}_{\texttt{IC.dest}}\mathcal{G}_{\texttt{score=sum(T)}}(\texttt{SC} \bowtie_\phi \texttt{IC})) \\ \oplus_{\texttt{score}} \\ \pi_{\texttt{IC.dest, SC.score}}({}_{\texttt{IC.dest}}\mathcal{G}_{\texttt{score=sum(T)}}(\Delta_{\texttt{SC}} \bowtie_\phi \texttt{IC}))\end{array}$$

Since the binary operator $\oplus_{\texttt{score}}$ is distributed by the relational algebra operators used in this query, we can incrementalize this query. The incrementalized query is shown in Figure 8, where `dScore` corresponds to a delta table of `Score`. Note that we use the termination condition as the optional filtering function ψ (line 34) described above. □

Groupby distribution over tuple insertion/deletion

Incremental computation of aggregation functions used in the group-by operation is worth further investigation for improving efficiency. While Figure 7 shows distribution rules for tuple updates, we show below distribution rules for dealing with tuple insertion/deletion when aggregation functions are used in the group-by operation.

For the aggregate function sum. we have the following distribution rules.

$$\begin{array}{l}{}_k\mathcal{G}_{sum(m)}\,(T \uplus \Delta_T) \\ \quad = \ {}_k\mathcal{G}_{sum(m)}\,(T) +_m \ {}_k\mathcal{G}_{sum(m)}\,(\Delta_T) \\ {}_k\mathcal{G}_{sum(m)}\,(T \setminus \Delta_T) \\ \quad = \ {}_k\mathcal{G}_{sum(m)}\,(T) -_m \ {}_k\mathcal{G}_{sum(m)}\,(\Delta_T)\end{array}$$

The first distribution rule says that if tuples Δ_T are inserted into T, the group-by operation ${}_k\mathcal{G}_{sum(m)}\,(T \uplus \Delta_T)$ can be computed

```
Schema:
Graph(src,dest,score)
Count(src,count)
Score(dest,score)
IT(src,dest)
IT_Count(src,dest,count)
Score'(dest,score)
dScore(dest,score)

Query:
 1: initialize
 2:   IT =
 3:     select src,dest from Graph;
 4:   Count =
 5:     select src,count(dest)
 6:     from IT group by src;
 7:   IT_Count =
 8:     select IT.src,IT.dest,Count.count
 9:     from IT, Count
10:     where IT.src = Count.src;
11:   Score' =
12:     select ic.dest,sum(sc.score/ic.count) as score
13:     from Score as sc, IT_Count as ic
14:     where sc.dest = ic.src
15:     group by ic.dest;
16:   dScore =
17:     select dest,(Score'.score - Score.score) as score
18:     from Score, Score'
19:     where Score'.dest=Score.dest
20:     and not(|Score'.score-Score.score| < ε);
21: iterate
22:   set Score =
23:     select dest,(Score.score + dScore.score)
24:     from Score, dScore
25:     where Score.dest=dScore.dest;
26:   set dScore =
27:     select ic.dest,sum(sc.score/ic.count) as score
28:     from dScore as sc, IT_Count as ic
29:     where sc.dest = ic.src
30:     group by ic.dest;
31:   set dScore =
32:     select *
33:     from dScore
34:     where not(|score| < ε);
35: until |new.score - old.score| < ε on Score;
36: return Score;
```

Figure 8: Incrementalized PageRank Query

incrementally by adding ${}_k\mathcal{G}_{sum(m)}\,(\Delta_T)$ to the original result $({}_k\mathcal{G}_{sum(m)}\,(T))$. The second equation is for the case when a set of tuples is deleted.

There are similar distribution rules for *count* omitted here. For the aggregation function *average*, it can be defined in terms of *sum* and *count*, so its incremental computation can be realized through those of *sum* and *count*.

For the aggregation functions *max* and *min*, we have the following distribution rules for tuple insertion.

$$\begin{array}{l}{}_k\mathcal{G}_{min(m)}\,(T \uplus \Delta_T) = \\ \quad {}_k\mathcal{G}_{min(m)}\,(T) \ \mathbf{min}_m \ {}_k\mathcal{G}_{min(m)}\,(\Delta_T) \\ {}_k\mathcal{G}_{max(m)}\,(T \uplus \Delta_T) = \\ \quad {}_k\mathcal{G}_{max(m)}\,(T) \ \mathbf{max}_m \ {}_k\mathcal{G}_{max(m)}\,(\Delta_T).\end{array}$$

However, attention should be paid to the case of tuple deletion; ${}_k\mathcal{G}_{min(m)}\,(T \setminus \Delta_T)$ (or ${}_k\mathcal{G}_{max(m)}\,(T \setminus \Delta_T)$) can be computed incrementally only if the minimum (or maximum) value in Δ_T is not equal to that in T.

EXAMPLE 3.2 (TOWARDS INCREMENTAL k-MEANS). As a simple example, consider the following query pattern (a group-by

after a selection):

$$_k\mathcal{G}_{sum(m)}\,(\sigma^*_{n:=f(n,T_1)}(T_2))$$

where $\sigma^*_{a:=v}(R)$ is introduced as an extension of the standard selection operation in that if the value of attribute a equals v, the tuple will be selected, otherwise the tuple will be updated with v being the value for the attribute a. We can apply the above distribution rules to incrementalize the query with the observation that

$$\sigma^*_{n:=f(n,T_1)}(T_2)$$

is equivalent in form to tuple insertion and deletion:

$$\mathtt{T}_2 \uplus \Delta_{\mathtt{T}_+} \setminus \Delta_{\mathtt{T}_-}.$$

This observation is based on the following decomposition rule

$$\sigma^*_{a:=v}(T) = T \uplus \sigma^*_{a:=v}(T') \setminus T'$$

where $T' = \sigma_{a\neq v}(T)$ consists of all tuples (in T) with their a values changed.

Note that we can realize incrementalization of the k-means query similarly. Recall the query for k-means clustering in Figure 6. We may start by rewriting the body of the step query in terms of relational algebra. By defining `cidF` as

```
cidF(p,Centroid) =
          select c.id
          from Centroid as c
          order by distance(c,p)
          limit 1
```

we can turn the step query into the following:

$$\textbf{set}\ \text{Centroid} = {}_{\mathtt{cid}}\mathcal{G}_{\mathtt{avg(pos)}}\,(\sigma^*_{\mathtt{cid:=cidF(p,Centroid)}}\,(\mathtt{Point}))$$

which is similar in form to what we saw at the beginning of the example, and can be automatically incrementalized. □

4. MAPREDUCE IMPLEMENTATION

We extend Hive [27], a query engine built on MapReduce, to evaluate iterative queries with convergence conditions and to integrate it with the query optimization of OptIQ.

The Hive query compiler is extended to parse queries written in the syntax for iterative queries in Figure 2 and to produce iterative query plans with convergence tests. For OptIQ extension, the query compiler extracts invariant and variant views and produces query plans that materialize the invariant views before iterations and that incrementally evaluate the variant view.

4.1 Iterative query processing

A query is compiled into a query plan consisting of a sequence of MapReduce jobs. Expressions in initialize clause, iterate clause, and return clause are compiled into MapReduce jobs. Step queries in iterate clause are repeatedly evaluated until convergence. The convergence condition is expressed by comparing new tuples and old tuples of update table and is tested at the end of every iteration. The comparison is made by joining the update table in the previous iteration with that in the current iteration, since the new and old tuples are related by the key. The new tuples are kept in a DFS (distributed file system) in each iteration, and then read as old tuples in the next iteration. Remember that there are two types of step queries specified by let/set statements. Since tables specified by the set statement are iteration global, they are also stored in DFS and read from MapReduce jobs in the next iteration.

4.2 View materialization

We extend the query compiler by applying the view materialization technique in Section 3.1; the queries for materializing invariant views are extracted and the materialized views are put on DFS and reused in subsequent iterations.

In MapReduce level, we also optimize the partition key of invariant views for efficient group-by and/or join operations on MapReduce. If invariant views are used for group-by/join operations in iterations, we partition the views by the key of the group-by/join operations, so that the group-by/join operations are computed in map tasks by applying map-side join [16]. The map-side join is a join technique on MapReduce that makes joins without shuffle by pre-partitioning the join tables by the join key. We can also apply a similar technique to group-by operations.

In the PageRank example in Figure 5, `IT_CNT` is materialized and partitioned by `src`. The join between `IT_CNT` and `Score` by `src` is efficiently made by applying map-side join.

4.3 Incrementalization

We apply the automatic incrementalization technique in Section 3.2 to the query compiler. The implementation consists of three parts; 1) the query compiler generates incremental query plans when the operators are distributive, otherwise it generates non-incremental query plans. 2) incremental query plans take delta tables as input and they output both delta tables and original tables. 3) delta tables are stored in DFS so they can be accessed from different iterations.

As we have seen in Section 3.2, the aggregation functions may not be incrementally computed, so the query compiler generates query plans, trees of operators, depending on whether the operators are incrementally computed. When the operators are distributive, they are incrementally computed by inputting delta tables and outputting both the original tables and their delta tables. Otherwise the operators are computed in non-incremental manner. At the MapReduce level, delta tables kept on DFS are obtained as follows. In the first iteration, delta tables are obtained by joining the old and new update tables and comparing new and old values of the modified attributes. Remember that the modified attributes are detected at table decomposition. After the first iteration, delta tables are obtained as the result of executing delta-based query plans by inputting the delta tables of the previous iteration.

In the PageRank example in Figure 5, `score` is a modified attribute so the delta table for $\mathtt{Score}(\mathtt{dest},\mathtt{score})$ is extracted by comparing the old and new values of `Score` table. Since the convergence condition computes the difference between old and new values of `Score`, the delta table extraction is optimized by being shared with the convergence test. The query plan inputs the delta table of `Score` and outputs both `Score` and its delta table.

The incremental computation of the k-means clustering in Figure 6 works as follows. In the first iteration, `ΔCentroid` is extracted by comparing the old and new values of modified attribute `pos` in `Centroid`. After the first iteration, `ΔVT` is obtained by inputting `ΔCentroid` in the previous iteration to the first step query. Then the new `ΔCentroid` is obtained by inputting `ΔVT` to the second step query. Notice that the inner query of the first step query, which locates the centroid in `Centroid` that is the closest data point in `IT`, is rewritten into a query by using **min** operation. That is

$$_{IT.id}\mathcal{G}_{min(distance(c.pos,IT.pos))}\,(Centroid).$$

Then, this query is incrementally evaluated except the following case. For every tuple in IT, if $\Delta Centroid$ is removed from $Centroid$ and the minimum value in the previous iteration is obtained from a centroid in $\Delta Centroid$, the above query is not

incrementally computed and the query executer falls back to non-incremental query processing.

4.4 Other optimizations

In addition to the query optimization of OptIQ, we employ design patterns for SQL processing on MapReduce, map-side join and memory-backed join in [16], and generate efficient MapReduce jobs.

For the PageRank in Figure 5, the iterate clause of the query is compiled into a single MapReduce job. As described before, `IT_CNT` is materialized and partitioned by `src` so the join between `IT_CNT` and `Score` by `src` is efficiently made by map-side join. The join result is shuffled by `ic.dest` for group-by operation. The reduce tasks computes the group-by operation and tests the convergence condition. The shuffled result is aggregated to compute `sum(sc.score/ic.count)` generating new `Score` and, then it is joined with the old `Score` for the convergence test. This join is computed only in the reduce tasks by employing the same idea of the map-side join, since the new and old `Score` are partitioned with the same key.

The iterate clause of the k-means clustering in Figure 6, which consists of two step queries, is implemented in a single MapReduce job. The first step query is compiled and optimized to apply memory-backed join to every tuple in `IT` to locate the closest centroid in `Centroid`, when the number of clusters are small. This optimization is efficient because `Centroid` can be kept in memory of map tasks. Then, data points are shuffled by the closest centroid for the group-by operation in the second step query; this aggregates a new position for each centroid in reduce tasks.

5. EXPERIMENTS

We demonstrate the effectiveness of OptIQ by removing redundant computations from the running examples, PageRank computation and k-means clustering. We compare the performance of OptIQ and Spark [29]. In addition, to avoid the overhead of accessing DFS between iterations in MapReduce, we also validate the efficiency of the view materialization and the incrementalization techniques implemented on top of Spark without implementing the query compiler: we manually implement Spark version programs for the running examples.

5.1 Hardware and Software Settings

The experiments were made on an 11-node cluster, a single master and 10 worker nodes. Each node has a single 2.80 GHz Intel Core 2 Duo processor running 64-bit CentOS 5.6 with 8GB RAM and 1TB SATA hard disk. According to hdparm, the hard disks deliver 3.7GB/sec for cached reads and 97MB/sec for buffered reads. The machine nodes are connected via a Fujitsu SR-S324 switch.

We implemented OptIQ on Hive 0.9 with Hadoop 1.0.3 and Oracle JDK 1.6.33. We used Spark 0.7.0. Hadoop uses a central job tracker and a master daemon for a distributed file system (HDFS) to coordinate node activities. To ensure that these daemons did not affect the performance of worker nodes, we executed both of these additional framework components on a master node in the cluster. We deployed the system using the default configuration settings, except for the changes indicated below by following the performance tuning method written in [22]: 1) the sort buffer size is set to 256MB instead of the default 100MB, 2) the JobTracker ran on JVM with maximum heap size of 4096MB, and the NameNode/-DataNode/TaskTracker ran on JVMs with maximum heap size of 1024MB. In addition, we configured the system to run two map tasks and two reduce tasks concurrently on each node, so as to efficiently use the dual cores of the nodes. For Spark, we configured the system to work similar to Hadoop, but set the memory size per machine to 6GB. The distributed file system used HDFS to store all input and output data. We also followed the performance tuning method written in [22]: 1) data is stored using 256 MB data blocks instead of the default 64MB, 2) with single replica, and 3) without compression. We used the sequence file format, which is more efficient than the text file format.

Table 1: Statistics of datasets

dataset	data type	# of nodes	# of edges	file size
wikimedia	graph	9.4M	215M	2.05GB
webbase-2001	graph	118M	1020M	18.1GB

dataset	data type	# of vectors	vector length	file size
US Census	vector	2.5M	68	0.7GB
mnist8m	vector	8.1M	784	16.8GB

5.2 Workload

We used iterative queries of PageRank and k-means clustering. We also made experiments on RWR queries and found that the results were similar to those of PageRank, so we omit them in this paper. The damping factor for PageRank is set to 0.85 and ε is set to 1% of the initial PageRank node score[3]. For k-means clustering, the number of clusters, k, is set to 500 and ε is set to 0.1[4].

We used four datasets, two graph datasets for PageRank and two vector datasets for k-means clustering. The statistics are shown in Table 1. The graph datasets are wikimedia dump on 20120601[5] and webbase-2001, a web graph[6] crawled by Stanford Webbase project. The vector datasets are US Census Data available at UCI machine learning repository[7] and mnist8m, handwritten digits data, which is the largest dataset in LIBSVM data collection[8].

5.3 Results

We measured the total response time, the time elapsed for each iteration, and the number of converged data over iterations using the above workload. The total response time is the response time of each whole query. To evaluate the effectiveness of the view materialization and incremental view evaluation, we used three settings, *default*, *view*, and *view+incremental* both on MapReduce and Spark. The default setting is without OptIQ, the view setting is with the view materialization technique of OptIQ, and view+incremental setting is with both the view materialization and incrementalization techniques of OptIQ. We utilize the filtering function in the incrementalization technique.

5.3.1 PageRank

Figure 9 shows the results on the graph datasets, wikimedia and webbase-2001. Overall, the incremental evaluation contributes significantly to the total response time on MapReduce and Spark as depicted in (a) and (d) in the figure. The combination of view materialization and incremental evaluation improved the total response time by up to five-fold compared to the default setting. In detail,

[3]Both are the same settings used in REX [19].

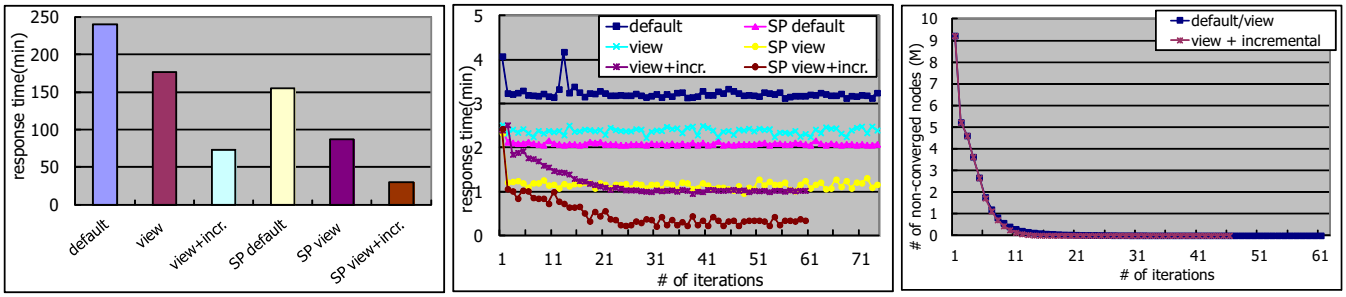
[4]ε=0.1 is small relative to the diameter of the vector space; 74K in US Census and 38M in mnist8m datasets.

[5]http://dumps.wikimedia.org/enwiki/20120601

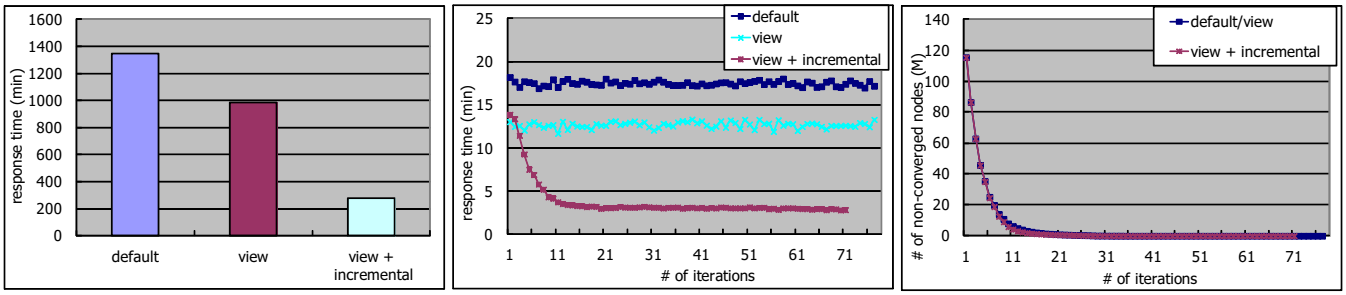
[6]http://law.di.unimi.it/datasets.php

[7]http://archive.ics.uci.edu/ml

[8]http://www.csie.ntu.edu.tw/~cjlin/libsvm



(a) wikimedia: total response time (b) wikimedia: iteration time (c) wikimedia: convergence



(d) webbase-2001: total response time (e) webbase-2001: iteration time (f) webbase-2001: convergence

Figure 9: PageRank results (SP and incr. indicate Spark and incremental, respectively.)

(b) and (e) depict the response time over iterations. As expected, the response time of the incremental evaluation decreases with the number of non-converged nodes as shown in (c) and (f). The view materialization without incremental evaluation reduced the total response time by around 30%. Note, Spark runs faster than MapReduce for wikimedia because Spark keeps data in-memory so DFS is not accessed between iterations. However, it runs out of memory in webbase-2001 since the data size of webbase is larger than wikimedia. We also found that the combination of view materialization and incremental evaluation on MapReduce is faster than the default setting of Spark.

Note that the incremental setting requires fewer iterations to convergence than the other settings. This is caused by the filtering function that changes the accuracy of the result as we discussed in Section 3.2.3. The filtering function additionally tests the convergence on every node and if the score of the node satisfies the convergence condition, it does not distribute its delta scores to the neighbors. In contrast, the default setting always distributes the scores of all nodes to the neighbors, so it requires more iterations. How much error is caused by the filtering function? We measured the average error rate of all nodes and found that it was close to ε (Remember ε is set to 1% of initial node score); 1.3% in webbase-2001 and 0.60% in wikimedia to the correct scores. We also conducted experiences where ε was set to 0.1% of the initial node score. The result showed that the response time of the incremental evaluation was slightly worsened (less than 3%), while the average error rate is improved five-fold to 0.20% in webbase-2001 and 0.16% in wikimedia. In addition, we evaluated ranking errors caused by changing ε both in default and incremental settings. We measured the ranking error of the top-1000 nodes when ε was set to 0.1% of the initial node score by assuming that the obtained ranking is correct when ε is set to 0.01% of the initial node score in the default setting. The ranking error is computed by using Spearman's rank correlation coefficient. The result showed that the default and incremental settings provide seven nines and five nines of preciseness, respectively against wikimedia, and six nines and three nines of preciseness, respectively against webbase-2001. This result indicates that it is impossible to remove errors in PageRank computations by the iterative method. The incremental technique provides users with high efficiency while permitting minor errors.

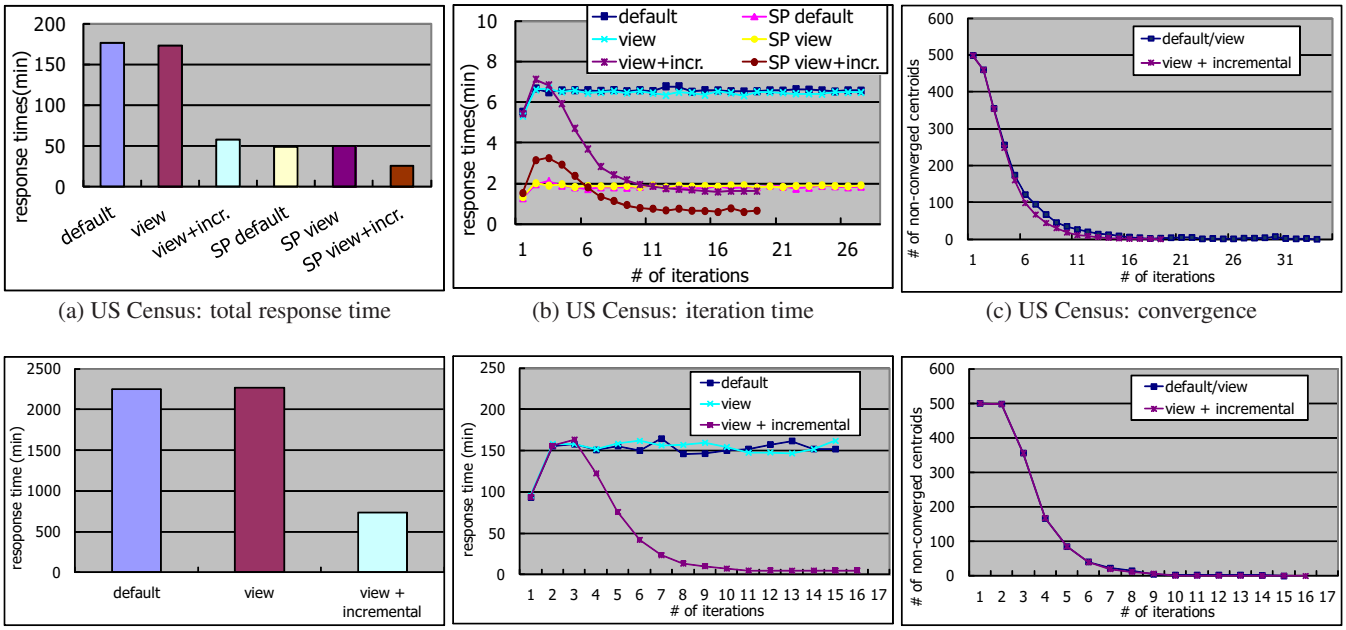
5.3.2 *k-means clustering*

Figure 10 shows the results on the vector datasets, US Census and mnist8m, where the number of clusters, k = 500. Overall, the results are similar to PageRank except the view setting. The incremental evaluation significantly contributes to the total response time as depicted in (a) and (d) in the figure; the total response time was less than that of the default setting in both datasets. The response time of the incremental evaluation in (b) and (e) decreased with the number of non-converged nodes as shown in (c) and (f). Spark runs three times faster than MapReduce on US Census but runs out of memory on mnist8m due to the large data size. The incremental setting requires fewer iterations to convergence than the default setting. The reason is the same as for the PageRank example. The average error of centroids was very small; it was 2.1 and 1.2 in US Census and mnist8m datasets, respectively, while the diameter of the vector space was 74K and 38M for each.

We observed that the view setting does not improve the response time compared to the default setting. We found that even if the view materialization removes redundant computations by minimizing the write IO cost, it may increase the read IO cost. Compare the original query for k-means clustering in Example 2.2 and the optimized query in Figure 6. The write IO cost in the optimized query is minimized by removing the redundant update on `Point'.pos` attribute, however an additional join is required between `VT` and `IT` in the second step query. Future work includes applying cost-based query optimization to obtain more efficient queries with minimized read/write IO cost.

5.4 Applicability of incrementalization

Incrementalization is applicable to queries written in our language when the queries are distributed, that is, the distribution rules



(a) US Census: total response time (b) US Census: iteration time (c) US Census: convergence

(d) mnist8m: total response time (e) mnist8m: iteration time (f) mnist8m: convergence

Figure 10: k-means clustering results (k=500. SP and incr. indicate Spark and incremental, respectively.)

shown in Figure 7 are applicable to the queries. Actually, our language is powerful enough to implement the variations of PageRank algorithms (personalized PageRank and RWR) and iterative clustering algorithms such as k-means clustering and NMF. The effectiveness of incrementalization depends on the queries and input datasets. For PageRank, the benefit of incrementalization comes from the fact that PageRank scores of most nodes converge early. This is caused by the power-law distribution found in real datasets, web and social graphs [11]. For k-means, vector datasets in the real world usually have large dependencies between dimensions and there are separate clusters in different size. Since the change of centroids has a local effect (only to neighbor clusters), the majority of clusters tend to converge in early iterations. Furthermore, for NMF, one of popular methods is the multiplicative update rule [14] that iteratively updates two matrixes and the computation can be expressed in our query language. NMF also benefits from incrementalization; most elements of the two matrixes obtained from real datasets converge early because of the power-law distribution.

6. RELATED WORK

This work is unique in its novel language-based approach (program analysis and program transformation) to fully automatic optimization of high-level iterative queries. It is strongly related to the work on runtime extensions and new language constructs for supporting iterative MapReduce processing, and was inspired by the success of declarative languages and optimization frameworks for MapReduce programming.

Iterative MapReduce Runtime Systems

There has been lots of work on iterative MapReduce runtime systems for distributed memory architectures. For instance, Twister [6], together with its extension Twister4Azure [23], is an enhanced MapReduce runtime with an extended programming model that supports iterative MapReduce computations efficiently. It supports map/reduce tasks operating with both static and variable data by introducing the *configure* phase to guide the map and reduce the tasks that load (read) any static data while avoiding the necessity of reloading static data in each iteration. Different from these lower-level runtime support tools, our work focuses on a *high-level* declarative programming framework for the automatic construction of efficient iterative MapReduce programs.

Iterative MapReduce Programming Models

HaLoop [3] provides a set of new APIs for dealing with data caching explicitly. It realizes the join of static data and state data with an additional MapReduce job, and implements a flexible task scheduler and caching techniques to maintain local access to static data. In contrast to our work, HaLoop is not intended as a high-level declarative language for expressing iterative queries; rather, it focuses on efficient basic APIs for iterative MapReduce programs.

iMapReduce [30] proposes the unique concept of persistent tasks to avoid repeated task scheduling, and facilitates asynchronous execution of map tasks within the same iteration to break the synchronization barrier among MapReduce jobs. Spark [29] proposes the concept of resilient distributed datasets (RDDs), which is a read-only collection of objects maintained in memory across iterations that supports fault recovery. Unlike iMapReduce and Spark, which require explicit specification of persistent tasks or resilient distributed data, we make full use of the techniques of program analysis and program transformation to automatically detect static data and persistent tasks (and thus simplifying iterative MapReduce programming.)

To ease the automatic parallelization of iterative computations on large-scale graphs, some graph-parallel abstractions (specific computation patterns) have been introduced to encode computation as vertex-programs that run in parallel and interact with each other. Examples include the bulk synchronous message passing style of Pregel [18], the GAS (gather, apply, and scatter) computation pattern of GraphLab [17], and matrix multiplication of PEGASUS [12]. In comparison, our automatic parallelization method targets arbitrary queries and aims to deal with general computations on large-scale graphs.

Declarative MapReduce Programming

Our work was greatly inspired by the success of many studies on high-level languages for making MapReduce programming easy. HiveQL [27] provides a high-level query language that allows users to write declarative queries, which are optimized and translated into MapReduce jobs that are executed using Hadoop. Pig [8] resembles Hive as it provides a user-friendly query-like language, called PigLatin [20], on top of MapReduce. HadoopDB [1] adopts a hybrid scheme between MapReduce and parallel databases to gain the benefit of both systems. It is interesting to see that an automatic optimization framework can be made to support general SQL-like map-reduce queries [7] in MRQL. Unfortunately, none of them support iterative MapReduce processing or its optimization, which motivated us to investigate to what extent we may extend these high level language frameworks (with effective optimization) to iterative MapReduce computation.

Query Optimization in MapReduce

Comet [10] is a cost-based optimizer which shares computations to remove redundancies both at the SQL level (shared SQL operations) and at the MapReduce level (shared scan and shuffling). YSmart [15] is a rule-based optimizer that exploits correlations among input tables and operators that have the same partition key. All of them are general query optimizers that pay no special attention to iterative queries.

For incrementalization, REX [19] explicitly handles programmable deltas, where changes can be propagated from iteration to iteration. However, programmable deltas rely on explicit designation of table deltas, and require users to write SQL or Java code to handle change propagation. Our incrementalization is fully automatic.

7. CONCLUSION

We proposed OptIQ, a fully automatic query optimization approach for iterative queries with convergence property. OptIQ removes redundant computations among different iterations and consists of two techniques. The view materialization technique removes redundant computations on unmodified attributes among iterations by introducing table decomposition and query lifting. The incremental evaluation technique removes redundant computations on unmodified tuples. We formalized delta table extraction and derived incremental evaluation for iterative queries. In addition, we described how iterative queries are compiled and OptIQ is implemented in the MapReduce environment. Experiments on real datasets showed that OptIQ achieves high efficiency.

8. 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, Aug. 2009.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

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

[5] J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[6] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *Proc. HPDC*, pages 810–818, 2010.

[7] L. Fegaras, C. Li, and U. Gupta. An optimization framework for Map-Reduce queries. In *Proc. EDBT*, pages 26–37, 2012.

[8] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proc. VLDB Endow.*, 2(2):1414–1425, Aug. 2009.

[9] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD*, pages 157–166, 1993.

[10] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *Proc. SoCC*, pages 63–74, 2010.

[11] S. Kamvar, T. Haveliwala, and G. Golub. Adaptive methods for the computation of pagerank. *Linear Algebra and its Applications*, 386:51–65, July 2004.

[12] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Proc. ICDM*, pages 229–238, 2009.

[13] C. Koch. Incremental query evaluation in a ring of databases. In *Proc. PODS*, pages 87–98, 2010.

[14] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *In NIPS*, pages 556–562, 2001.

[15] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet another SQL-to-MapReduce translator. In *Proc. ICDCS*, pages 25–36, 2011.

[16] J. Lin and C. Dyer. *Data intensive text processing with MapReduce*. Morgan & Claypool, 2010.

[17] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. SIGMOD*, pages 135–146, 2010.

[19] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: Recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.

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

[21] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: massively parallel learning of tree ensembles with MapReduce. *PVLDB*, 2(2):1426–1437, 2009.

[22] 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 *Proc. SIGMOD*, pages 165–178, 2009.

[23] J. Qiu, T. Gunarathne, and G. Fox. Classical and iterative mapreduce on AzureClassical and iterative MapReduce on Azure. In *Cloud Futures 2011 workshop*, 06/2011 2011.

[24] K. Shim. MapReduce algorithms for big data analysis. *PVLDB*, 5(12):2016–2017, 2012.

[25] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In *Proc. NIPS*, pages 281–288, 2007.

[26] The Apache Software Foundation. Mahout. http://mahout.apache.org.

[27] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *Proc. ICDE*, pages 996–1005, 2010.

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, pages 2–2, 2012.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proc. HotCloud*, pages 10–10, 2010.

[30] Y. Zhang, Q. Gao, L. Gao, and C. Wang. iMapReduce: A distributed computing framework for iterative computation. In *Proc. IPDPSW*, pages 1112–1121, 2011.