Optimizing Coordinate Descent over Normalized Data in Column Stores

ABSTRACT

1. INTRODUCTION

ML Technique	$F_e(a, b)$ (For Loss)	$G(a, b)$ (For Gradient)
Logistic regression (**LR**)	$\log(1 + e^{-ab})$	$\frac{-a}{1 + e^{ab}}$
Least-Squares Regression (**LSR**), Lasso, and Ridge	$(a - b)^2$	$2(b - a)$
Linear Support Vector Machine (**LSVM**)	$\max\{0, 1-ab\}$	$-a\delta_{ab<1}$

Table 1: GLMs and their functions.

2. BACKGROUND AND PRELIMINARIES

We provide a brief introduction to GLMs, CD and MonetDB. For a deeper understanding and interests, we refer the readers to [**?**][**?**][**?**][**?**], we use the same notation used in [**?**].

2.1 Generalized Linear Models (GLMs)

Consider a matrix $\mathbf{X}$ with dimension $n \times d$ representing a dataset including n samples and each sample has d features. Then the i_{th} row of matrix $\mathbf{X}$(namely $\mathbf{X}_{(i,)}$) represents the i_{th} sample in the dataset with its corresponding d features; the j_{th} column of matrix $\mathbf{X}$(namely $\mathbf{X}_{(,j)}$) represents the j_{th} feature of all examples in the dataset. Y is a n-dimensional vector representing numerical targets for all samples in the datasets. For instance, Y_i is a numerical target for $\mathbf{X}_{(i,)}$. $Y_i \in \mathbb{R}$ considering regression (continuous target values). For discrete target values, however, Y_i represents the numerical value representing the defined class, such as classification in binary class, $Y_i \in \{-1, 1\}$. GLMs make the assumption that the data in dataset could be distributed in different classes (discrete target values for classification) or assigned approximated values (continuous target values for regression) by a hyperplane. Such hyperplane is called "model", and the ultimate goal is to use the given dataset $\mathbf{X}$ to compute the model $\mathbf{W} \in \mathbb{R}^d$. To evaluate how accurate the model is, a *linearly-separable* objective function is given to compute the *loss* of the model $\mathbf{W} \in \mathbb{R}^d$ on the data: $F(\mathbf{W}) = \sum_{i=1}^{n} F_e(Y_i, \mathbf{W}^T \mathbf{X}_{(i,)}^T)$. Applying proper ML algorithm to find the optimal solution for model $\mathbf{W} \in \mathbb{R}^d$ to minimize the loss function is the ultimate goal. The corresponding mathematical statement is: find a vector $\mathbf{W}^* \in \mathbb{R}^d$ s.t., $\mathbf{W}^* = argmin_{\mathbf{W}^*} F(\mathbf{W})$.

2.2 Coordinate Descent(CD)

Coordinate Descent(CD) is a column-friendly algorithm (fetch one column each time for a single coordinate update), thus we consider CD in column stores taking advantage of the fact that the data is stored in columns in the column-oriented DBMS (specifically we use MonetDB as paradigm in our work).

Here, we first just present the most basic version of CD for simplicity. The algorithm just updates one coordinate each time regarding other coordinates as fixed in each iteration. However, as noted in [**?**], most applications in real life use *block coordinate descent* (BCD), in which several coordinates regarded as one block are updated synchronously each time in each iteration regarding other "blocks" are fixed instead of updating just a single coordinate each time. We will briefly discuss BCD in the extensional work part and it shows that our new-designed techniques will also have better performance improvement (in terms of speed-up) in BCD compared to the *single coordinate descent* method.

Algorithm 1 Coordinate Descent (CD)

Inputs: $\{\mathbf{X}, Y\}$ (Data)

1: $k \leftarrow 0$, $r_{prev} \leftarrow$ null, $r_{curr} \leftarrow$ null, $H \leftarrow \mathbf{0}$, $\mathbf{W} \leftarrow \mathbf{0}$
2: **while** (Stop (k, r_{prev}, r_{curr}) = False) **do**
3: $\quad r_{prev} \leftarrow r_{curr}$
4: $\quad$ **for** j = 1 to d **do** $\quad \triangleright$ 1 pass over data
5: $\quad\quad \nabla F_j^k(\mathbf{W}) \leftarrow \sum_{i=1}^{n} G(Y_i, H_i) X_{(i,j)}$
6: $\quad\quad \mathbf{W}_j^{(k)} = \mathbf{W}_j^{(k-1)} - \alpha \nabla F_j^k(\mathbf{W})$
7: $\quad\quad H \leftarrow H + (\mathbf{W}_j^k - \mathbf{W}_j^{(k-1)}) \times \mathbf{X}_{(,j)}$
8: $\quad$ **end for**
9: $\quad r_{curr} \leftarrow F_k$
10: $\quad k \leftarrow k + 1$
11: **end while**

CD is a simple algorithm to solve GLMs using iterative numerical optimization. CD initializes the model $\mathbf{W}$ to some $\mathbf{W}^{(0)}$, considering a single coordinate every time in each iteration, the algorithm assumes all other coordinates are fixed except for $\mathbf{W}_j$, the j_{th} entry in the model to be updated. Compute the partial gradient $\nabla F_j(\mathbf{W})$ on the given dataset on coordinate W_j (corresponding to j_{th} feature) , where $\nabla F_j(\mathbf{W}) = \sum_{i=1}^{n} G(Y_i, \mathbf{W}^T \mathbf{X}_{(i,)}^T) \mathbf{X}_{(i,j)}$. And then updates the j_{th} entry in the model as $W_j \leftarrow W_j - \alpha \nabla F_j(\mathbf{W})$, where $\alpha > 0$ is the *learning rate* (stepsize) parameter. Once after finishing the update of current coordinate, the algorithm goes into the updating process for the next coordinate in the model $\mathbf{W}$. The coordinate descent that updates one single coordinate each time is also called *stochastic coordinate descent* (SCD). SCD is outlined in Algorithm 1.

SCD updates the model repeatedly, i.e., over many *iterations* (or *epoches*), each of which requires at least one pass of data. The loss value typically decreases over iterations. The algorithm stops typically with some pre-defined conditions (i.e., specific number of iterations, the decrease of loss value across iterations). The learning rate parameter (α) is typically selected using a line search method that potentially computes the loss many times. Conventionally, *re-ordering* (shuffling) of data in the dataset is done in every iteration to reduce the *influence* that might be caused by data ordering on learning process. However, many practical experiments show that one shuffling is good enough to mitigate the influence of data ordering, thus it is not necessary to shuffle the data in every iteration. We apply this idea in our implementations.

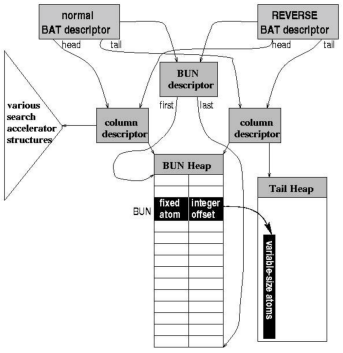
Here, we present the general version of CD algorithm which can be applied for all GLMs listed on Table 1. H is $n \times 1$ *residual vector*, which stores the $\mathbf{W}^T \mathbf{X}_{(i,)}^T$ for every $X_{(i,)}$ with current $\mathbf{W}$. In some cases, it is possible to find the optimal $\mathbf{W}_j$ mathematical expression. For reader's interests, we refer [**?**], which presents CD applied in LSR.

2.3 MonetDB

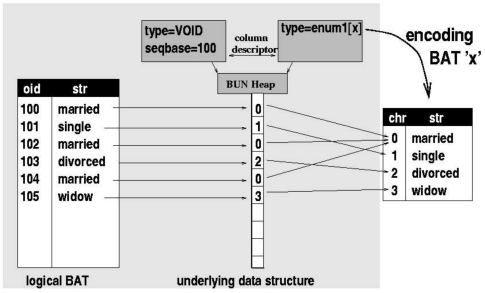
We use MonetDB, an open-source column store, as the paradigmatic column-oriented DBMS in this paper. In order to give readers a better understanding of our work, we provide a brief introduction to MonetDB. Readers can also refer to [**?**].

2.3.1 Data Structure

The data structure used in MonetDB is called *binary association table* (BAT). Every column in the table is stored in



(a) BAT Data Structure



(b) The logical mapping of BAT

Figure 1: MonetDB Data Structure

a BAT. The single unit in BAT is called *binary unit* (BUN). The two columns of BAT is called *BUN Heap* and the first column in BUN Heap is called *head*, which stores *objective identifier* (OID, used to refer the logical location of each tuple in the table); the second column in BUN heap is called *tail*, which stores the corresponding attribute (or feature). All BATs for the same table that store all different features have the same OID in the head columns in BUN Heaps. Figure 1 shows the graphic depiction of BAT.

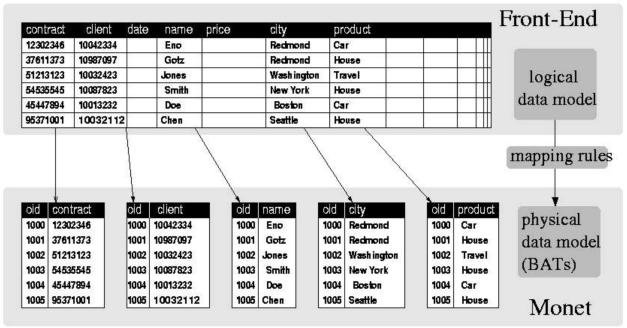
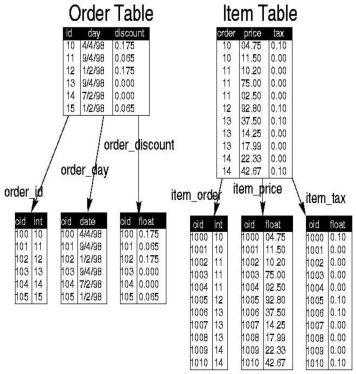


Figure 2: Examples of storage of relations in MonetDB

2.3.2 Storing Relations in MonetDB

In Figure 2, it shows how every attribute in the "contract" table is stored in the BAT data structure in MonetDB. We can observe that different attributes on the same row in the "contract" table have the same oid in different BATs, and this observation conforms to the description of BAT data structure given before. It should not be hard to conjecture that MonetDB uses oid to locate and fetch the data needed: once the oid for certain tuple in database is known, all attributes in this tuple could be fetched in corresponding BATs using the oid as reference. What should be noticed here is that for different tables, the oid generated by the database system will be different (length, sequence). In other words, oid is unique for different tables and **No** same oid can be or should be found in BATs for two different tables. Otherwise there will be conflicts and problems when fetching data in different tables but share the same oid.



(a) BAT mapping of "Order" and "Item"

oid	oid
1000	100
1001	100
1002	101
1003	101
1004	101
1005	102
1006	103
1007	103
1008	103
1009	104
1010	104

(b) oid-oid mapping

Figure 3: Relational mapping of "Order" table and "Item" Table

2.3.3 Examples of relational mapping

Figure 3 gives a general sense of relational mapping between two tables: let's say we want to join the "Order" table and "Item" table on order "id". then "id" would be the primary key of "Order" table and "order" in "Item" table is the foreign key refers to the order "id" in "Order" table. Looking at the order_id and item_order BATs in Figure 3(a), since the oids for all attributes are the same in the same tuple of the same table, once the oid-oid mapping of "Order" and "Item" is got, the join of "Order" table and "item" table can be completed by copying all the attributes to the corresponding position referred by oid following oid-oid mapping. The oid-oid mapping is shown in Figure 3(b). To get the oid-oid mapping, simple hash join is applied since only two BATs (primary key and foreign key) are needed (thus it is reasonable to assume that they can fit well in memory). We

Symbol	Meaning
$\mathbf{R}$	Attribute table
$\mathbf{S}$	Entity table
$\mathbf{T}$	Join result table
n_R	Number of rows in $\mathbf{R}$
n_S	Number of rows in $\mathbf{S}$
d_R	Number of features in $\mathbf{R}$
d_S	Number of features in $\mathbf{S}$ (includes Y)
p	Page size in bytes (1MB used)
m	Allocated buffer memory (pages)
f	Hash table fudge factor (1.4 used)
$\|\mathbf{R}\|$	Number of $\mathbf{R}$ pages ($\frac{8n_R(1+d_R)}{p}$)
$\|\mathbf{S}\|$	Number of $\mathbf{S}$ pages ($\frac{8n_S(2+d_S)}{p}$)
$\|\mathbf{T}\|$	Number of $\mathbf{T}$ pages ($\frac{8n_S(1+d_S+d_R)}{p}$)
$Iters$	Number of iterations of CD (≥ 1)

Table 2: Notation for objects and parameters used in the cost models. I/O costs are counted in number of pages. Dividing by the disk throughput yields the estimated runtimes. NB: As a simplifying assumption, we use an 8B representation for all values: IDs, target, and features (categorical features are assumed be have been converted to numeric ones [**?**]).

call this oid-oid mapping as "key-foreign key mapping reference"(KKMR) and will use this notation in later sections.

Problem Definition. Suppose there are n_S examples (tuples) in $\mathbf{S}$, and n_R tuples in $\mathbf{R}$. Assume that the feature vectors are split across $\mathbf{S}$ and $\mathbf{R}$, with $d_S - 1$ features in $\mathbf{X}_S$ and $d_R = d - d_S + 1$ in $\mathbf{X}_R$. Thus, the "width" of $\mathbf{S}$ is $2 + d_S$, including the ID, foreign key, and target. The width of $\mathbf{R}$ is $1 + d_R$, including the ID. Typically, we have $n_S \gg n_R$, similar to how fact tables have more tuples than dimension tables in OLAP [**?**][**?**]. We now state our problem formally.

Given two relations $\mathbf{S}$ $(\underline{SID}, Y, \boldsymbol{X}_S, FK)$ *and* $\mathbf{R}$ $(\underline{RID}, \boldsymbol{X}_R)$ *with a key-foreign key relationship (*$\mathbf{S}.FK$ *refers to* $\mathbf{R}.RID$*), where* $\boldsymbol{X}_S$ *and* $\boldsymbol{X}_R$ *are feature vectors and* Y *is the target, learn a GLM using CD over the result of the projected equi-join* $\mathbf{T}(\underline{SID}, Y, [\boldsymbol{X}_S \ \boldsymbol{X}_R]) \leftarrow \pi(\mathbf{R} \bowtie_{RID=FK} \mathbf{S})$ *such that the feature vector of a tuple in* $\mathbf{T}$ *is the concatenation of the feature vectors from the joining tuples of* $\mathbf{S}$ *and* $\mathbf{R}$*.*

3. SIMPLE APPROACHES

Assumptions and Cost Model. For the rest of the paper, the focus is only on $(\mathbf{W}, F)$, update of the model and computation of the total loss, which are the most computationally intensive parts and correspond to the 4-7 steps and 9 step in **Algorithm 1**. Similar to what is discussed in [**?**], we only focus on simple hash join ([**?**]) for join operation here. We focus primarily on the case $n_S > n_R$ and $|\mathbf{S}| > |\mathbf{R}|$. To specify the value of corresponding operations when estimating the CPU cost, we use the same notation as used in [**?**]

3.1 CD After a Join: Materialize (M)

In column stores, such as MonetDB, since the primary key in $\mathbf{R}$ and the foreign key in $\mathbf{S}$ can be fetched as two single columns (two BATs), the key-foreign key match needed in the join process thus could be completed in memory without causing extra I/O cost under the reasonable assumption that two single columns KKMR (oid-oid mapping) should be able to fit in main memory. Thus there is no partitioning step in the join operation in column stores such as in hybrid hash join in row stores, which will write the partitions to the disk and then read partitions and thus introduce extra I/O cost. In other words, the join operation in column stores is simply to apply "simple hash join" ([**?**]) on two columns that store primary key and foreign key to obtain KKMR, and then copy other entries in other columns to the corresponding positions following the "instructions" of KKMR. We now simply consider KKMR (oid-oid mapping), target column (Y), residual vector column (H), and any single column can fit in memory (this assumption is also used in MonetDB). To be compatible with "column wise", we introduce a few new notations. Assume that all columns in table R are of the same size and all columns in table S are of the same size, then we can denote the size of each column in R as $|Rcol|$ and $|Rcol| = \frac{|R|}{1+d_R}$; $|Scol| = \frac{|S|}{d_S+2}$. Correspondingly, $|T| = (d_S + d_R - 1)|Scol|$

Symbol	Meaning	Default Value (CPU Cycles)
hash	Hash a key	100
comp	Compare two keys	10
copy	copy a double	1
add	add two doubles	10
mult	Multiply two doubles	10
funcG	Compute G(a,b)	150
funcF	Compute Fe(a,b)	200

Table 3: Notation for the CPU cost model. The value of cost of funcG and funcF are specifically for LR.

I/O Cost

```
  (1+dS).|Scol|+(1+dR).|Rcol|
+ |T|
+ Iters.|T|
- min{|T|, (m-4|Scol|-f.|Rcol|)}
- min{|T|,(m-3|Scol|)}.(Iters-1)

If |T| < (m-4|Scol|-f|Rcol|):
  (1+dS).|Scol|+(1+dR).|Rcol|
+ |T|
+ |T|
```

CPU Cost

```
  nR.(copy+hash)
+ nS.(copy+hash+comp.f)
+ nR.dS.copy
+ Iters.[
   (dS+dR-1).[
      nS.funcG
    + nS.(add+mult)
    + (2.add+mult)
    + nS.(mult+add)
   ]
   + nS.(funcF+add)
  ]
```

3.2 CD Over a Join: Stream (S)

This approach differs from M in the way that S perform the join in "every" iteration but without writing **T** to the Disk. For the join operation,

1. Apply simple hash join to obtain **T**, but instead of writing **T**, compute (**W**,F) on the fly.
2. Repeat step 1 for every iteration

The I/O cost of *stream* is simply approximately the cost of the simple hash join multiplied by the number of iterations; the CPU cost is the combination of join and CD.

I/O Cost

```
  |Scol|
+ Iters.[
     dS.|Scol|+(1+dR).|Rcol|
 ]
- (Iters-1).[
     min{dS.|Scol|+(1+dR).|Rcol|,(m-4|Scol|-f|Rcol|)}
  ]
```

CPU Cost

```
Iters.[
  (nR+nS).hash
   +(nR + nS).copy
   +nS.comp.f
   +dR.[
         nS.copy
       + nS.funcG
       + nS.(add+mult)
       + (2add+mult)
       + nS.(add+mult)
     ]
     + (dS-1).[
         nS.funcG
       + nS.(add+mult)
       + (2add+mult)
       + nS.(add+mult)
      ]
       + nS.(funcF + add)
]
```

Redundancy Ratio. Consider the main case we focused: **S** is the *entity table* and **R** is the *attribute table* and $n_S \gg n_R$, when the key-foreign key join is processed between **R** and **S** on primary key of **R**, it is very likely that a single tuple in **S** will join multiple tuples in **R** (e.g., a same item can be placed on many different orders). Thus the materialized table **T** resulted from the join of **R** and **S** is usually larger than $|\mathbf{S}| + |\mathbf{R}|$. The techniques we discuss in this paper are all come up with the intention of reducing redundancy in either I/O or computation (or both) to improve the performance of learning on two tables. Thus, it would be helpful to introduce the term *redundancy ratio* for straightforward observation on factors that will influence the improvement of different approaches on the learning process. *Redundancy ratio* is defines as:

$$r = \frac{(d_R + d_S - 1)|Scol|}{d_R|Rcol| + (d_S - 1)|Scol|} = \frac{(d_S + d_R - 1)n_S}{d_R n_R + (d_S - 1)n_S}$$

$$= \frac{\frac{n_S}{n_R}(1 + \frac{d_R}{d_S} - \frac{1}{d_S})}{\frac{n_S}{n_R}(1 - \frac{1}{d_S}) + \frac{d_R}{d_S}}$$

The redundancy ratio is used in later analysis of the performance improvement for approaches discussed in this paper, mainly for *factorized learning*. We can observe that when $n_S >> n_R$, the redundancy ratio converges to $\frac{1+\frac{d_R}{d_S}-\frac{1}{d_S}}{1-\frac{1}{d_S}}$, and in this case, the redundancy ratio increases with the increase of $\frac{d_R}{d_S}$. Factorized learning speeds up by avoiding the redundant computation and I/O overhead result from join, thus the redundancy ratio could give a good sense of how much factorized learning can speeds up given specific n_S, n_R, d_S and d_R.

No Stream-Reuse (SR). [**?**] has introduced an approach called *stream-reuse*(SR): store the partitions for **R** and **S** to disk in the first iteration to avoid the both computational and I/O overhead of repetitive partitioning in later iterations. SR has taken advantage of the idea introduced in *Stream* and further reduces the I/O cost by avoiding writing partitions in every iteration and CPU cost by avoiding processing partitions of **R** and **S** in every iteration. In column stores such as MonetDB, since KKMR is used and can well fit in memory (only a single BAT), partition is not needed. Thus, there is **No** SR in column-store. However, we apply the similar idea in FL: calculate KKMR only once and keep it in memory to avoid re-calculation in later iterations.

4. FACTORIZED LEARNING (FL)

4.1 Mechanism of FL in Column-Store with CD

A new technique called *factorized learning*(FL) that interleaves the I/O and CPU process of the join and BGD has been introduced in [**?**], this paper has explored how FL can be applied in column stores with CD and compared the performance of FL in row stores with BGD and that in column stores with CD. For convenience of performance comparison and consistency, the notations used here are kept similar to that used in [**?**]. FL has been proved to perform very well in [**?**] with BGD in row stores in terms of speed improvement: in most cases, it is often (though not always) the fastest approach compared to M, S, and SR. The reason that FL has such good performance is that FL approach does not only avoids redundant I/O but also avoids the redundant computation of the inner product ($\mathbf{W}^T\mathbf{X}^T_{(i,)}$) caused by the duplicates $\mathbf{X}_{(i,R)}$ from **R** when constructing **T** based on the insight: $\mathbf{W}^T\mathbf{X}^T_{(i,)} = \mathbf{W}^T_S\mathbf{X}^T_{(i,S)} + \mathbf{W}^T_R\mathbf{X}^T_{(i,R)}$ (to keep consistency, here we use the "column-wise" notation as introduced in Section 2 before: $\mathbf{X}_{(i,)}$ is i_{th} feature vector representing the i_{th} sample in **T**). And the computation of $\mathbf{W}^T\mathbf{X}^T_{(i,)}$ is involved in both F and ∇F in BGD. However, in CD, since the model is updated after going through each single column (each coordinate is updated one by one asynchronously), the factorized technique based on the insight $\mathbf{W}^T\mathbf{X}^T_{(i,)} = \mathbf{W}^T_R\mathbf{X}^T_{(i,R)} + \mathbf{W}^T_S\mathbf{X}^T_{(i,R)}$ in BGD (in which every entry in the model could be updated synchronously) will not speeds up the performance if applied directly to CD. On the contrary, it will slow down the computational speed: considering **Algorithm 1**, the update of inner product stored in the residual vector **H** only involves the feature column corresponding to the coordinate

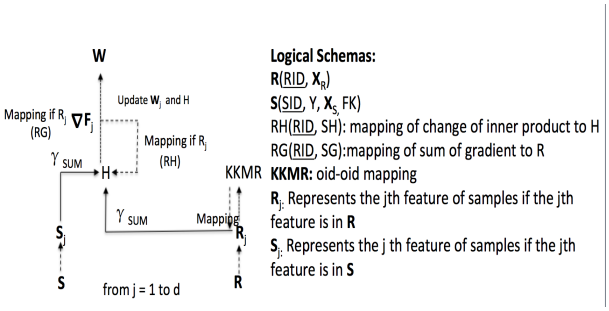


Figure 4: Complete logical workflow of factorized learning. Considering feature column S_j in **S**, simply calculate the partial gradient on corresponding column, and update the corresponding coordinate (W_j), and then update H. Considering feature column R_j in **R**, look up the KKMR (oid-oid mapping) both in computation of the partial gradient (mapping called "RG") and update of the residual vector H (mapping called "RH"). Here, γ_{SUM} denotes a `SUM` aggregation.

being updated, thus applying the "factorized" idea introduced in BGD will recalculate the full inner product and thus will introduce unnecessary computation involving irrelevant features, which will slow down the computation speed. There is a more *mathematical* factorized idea can be applied in CD: considering the step 5 and step 7 in **Algorithm 1** with corresponding columns in **R**, to form the complete feature column of R ($n_R \rightarrow n_S$) and then compute $\nabla F_j^k(\mathbf{W}) \leftarrow \sum_{i=1}^{n} G(Y_i, H_i) X_{(i,j)}$, the redundant computation is likely to be introduced: $G(Y_i, H_i)X_{(i,j)}$ with the same $\mathbf{X}_{(i,j)}$ being repeatedly computed (similar computational redundancy also appears in $H \leftarrow H + (\mathbf{W}_j^k - \mathbf{W}_j^{(k-1)}) \times \mathbf{X}_{(,j)}$). Specific example of gradient computation and update of H is given in Figure 5 and 6.

The complete logical workflow and the corresponding logical schema of FL are shown in Figure 4.

1. Before the iteration starts, compute KKMR (oid-oid mapping) with hash join and keep it in memory. Read column Y from **S** and keep in memory

2. When considering each feature column S_j in **S**, there is no extra work: the procedure just works as what is described in **Algorithm 1**. Compute the partial gradient ∇F_j by summing up the partial coordinate gradient on j_{th} feature of every sample. Update the corresponding coordinate $\mathbf{W}_j$, and then update the residual vector H, which stores the inner product of each $\mathbf{X}_i$, and current model **W**.

3. When considering each feature column R_j in **R**, to avoid the redundant computation, first map each partial gradient to every entry in **R** by looking up KKMR to get RG. After the mapping is done, compute the sum of corresponding partial gradient to every entry in RG to get the *full partial* gradient on R_j. Use the same way to get RH to avoid redundant computation of $\Delta W_j X_{(i,j)}$ on same $X_{(i,j)}$. And then update the H by looking at the KKMR again to find the corresponding entry in RH.

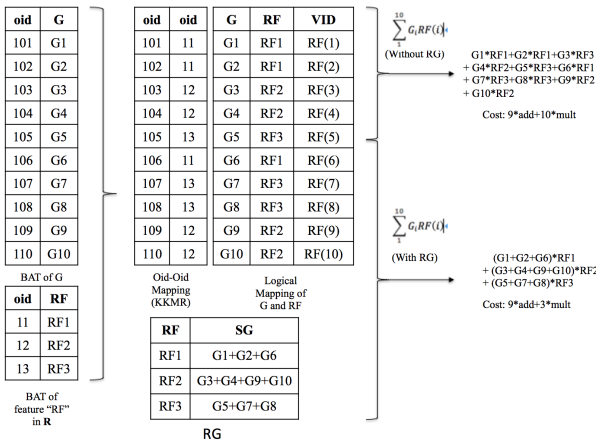


Figure 5: Considering a dataset with only ten samples (then the size of corresponding **S** is also ten), then the size of G is ten. Assume that the size of **R** is three, and one feature in **R** is **RF**, then "SG" for each entry RFj in **RF** is the sum of corresponding entries in G that map to RFj.

I/O Cost

```
2|Scol| + |Rcol|
+ Iters.[
(dS-1).|Scol| + (dR-1).|Rcol|
    ]
- (Iters-1).[
 min{(dS-1).|Scol|+dR.|Rcol|, m - 4S|col|}]
```

CPU Cost

```
  (nR+nS).hash
+(nR+nS).copy
+ f.nS.comp
+ Iters.[
    dR.[
    nS.funcG
 + nR.mult
 + nS.add
 + 2*add+mult
 + nR.mult
 + nS.add
]
    + (dS-1)[
            nS.funcG
         + nS.(add+mult)
         + 2*add+mult
         + nS.(add+mult)
        ]
        nS.(funcF+add)
   ]
```

5. CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LaTeX book, the citations in this paper are to articles

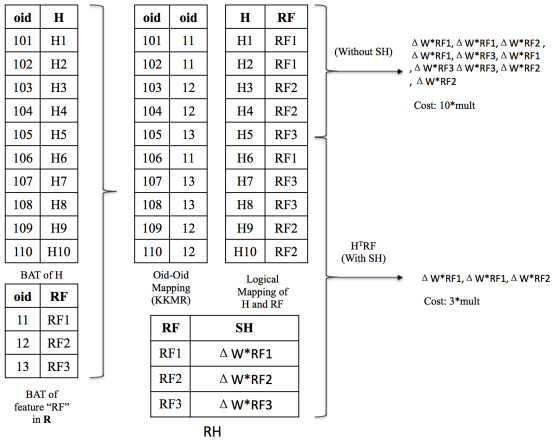


Figure 6: Considering a dataset with only ten samples (then the size of corresponding **S** is also ten), then the size of H is ten. Assume that the size of **R** is three, and one feature in **R** is **RF**, then "SH" for each entry RFj in **RF** is the change of partial inner product on $X_{(,j)}$ that maps to RFj.

which have nothing to do with the present subject and are used as examples only.

6. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

6.1 References

Generated by bibtex from your .bib file. Run latex, then bibtex, then latex twice (to resolve references).

APPENDIX

You can use an appendix for optional proofs or details of your evaluation which are not absolutely necessary to the core understanding of your paper.

A. FINAL THOUGHTS ON GOOD LAYOUT

Please use readable font sizes in the figures and graphs. Avoid tempering with the correct border values, and the spacing (and format) of both text and captions of the PVLDB format (e.g. captions are bold).

At the end, please check for an overall pleasant layout, e.g. by ensuring a readable and logical positioning of any floating figures and tables. Please also check for any line overflows, which are only allowed in extraordinary circumstances (such as wide formulas or URLs where a line wrap would be counterintuitive).

Use the `balance` package together with a `\balance` command at the end of your document to ensure that the last page has balanced (i.e. same length) columns.