Brown University

Technical Report

A Machine Learning Engine Using Iterative Transactions

Author:
Jin Yan

Supervisor:
Carsten Binnig
Tim Kraska

February 7, 2018

1 Architecture

1.1 Storage Schema

The storage schema is multi-version concurrent control (MVCC)[1] based. Each unit may store an array of multiple values, whose information is stored in meta data. This meta information contains current version, the corresponding index of current version in value array, and whether it is locked, which is shown in figure 1.

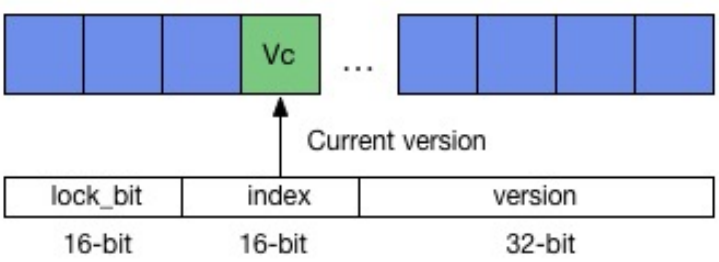


Figure 1: database architecture

As you can see, meta data for each sample is stored as a 64-bit integer. For the first 16 bits, only one bit is used to indicate whether it is locked. The current index takes 16 bits to store, and the current version takes the last 32 bits. The value array is a circular array, so the total length of value array is used to update index.

Read returns data and version. Our storage system allows one node to request a specific version of other nodes, or simply the most recent version. For example, in page rank algorithm, one node needs to calculate the new page rank value based on the same version of page rank values of incoming nodes. In that case, it must request a specific version of value from its neighbor. If its neighbor doesn't have it, the read function will return an error. On the other hand, an asynchronous ML algorithm always doesn't need the version check. In that case, the storage system will simply return the most recent version.

Write needs read version and distance. The read version refers to the version of value used to calculate the algorithm, and the distance refers to the difference between the read version and the current version of value. If this distance is bigger than a user-defined staleness boundary, the write operation will return an error. It is obvious that in the asynchronous mode, this check is unnecessary and the storage system will just write the value and update the current version.

1.2 Execution Model

Figure 2 shows the workflow of this engine, which is more like a database with iterative transactions.

[1] https://en.wikipedia.org/wiki/Multiversion_concurrency_control

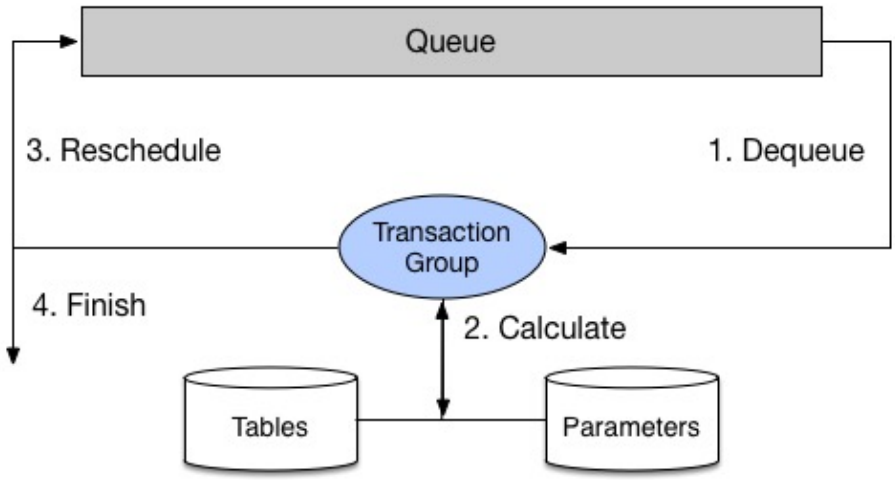


Figure 2: database architecture

The whole system consists of four components: queue, tables, executor, transactions. Each ML algorithm is split and assigned to a number of transactions, which will be calculated and rescheduled iteratively. Below we list principles for designation:

Queue-based execution: All the transactions are managed by a lock-free queue, in a first-in-first-out order. The database will initialize this queue at the very beginning, and all the executors will dequeue and inqueue transactions to it.

Non-preemptive thread scheduling: Whenever a core is free, it will fetch a transaction group from the queue, and calculate a piece of ML algorithm based on transaction programming interface, which is covered in section 2. If the transaction hasn't converged, the core will reschedule it and insert it into the queue.

Data localization: All critical hyper-parameters, as well as neighbor information needed for calculation, are cached in transaction as states. Some of the meta data will be updated during the process, so we also have function (`retrieve_state`) to get the most up-to-date information at the beginning of each pass of transaction.

Optimization1: Bulk Execution: In order to reduce the overhead of transaction scheduling, several transactions with or without relation will be wrapped together into one group, and they will be scheduled and executed together.

Optimization2: Repairing: Whenever a transaction fails to get the specific version of its neighbors, or fails to validate, the system will try to repair the nodes that cause this issue. In that case, the system will initiate a recursive call of transactions.

2 Programming Model

In order to facilitate runtime, we use curiously recurring template pattern (CRTP)[2], which allows the system to get rid of virtual table. The database engine only calls an abstract transaction that inherits from a template base class. Each algorithm has its own kind of transaction implementation. all these transactions can be the base class for the abstract one. This relationship is shown in figure 3:

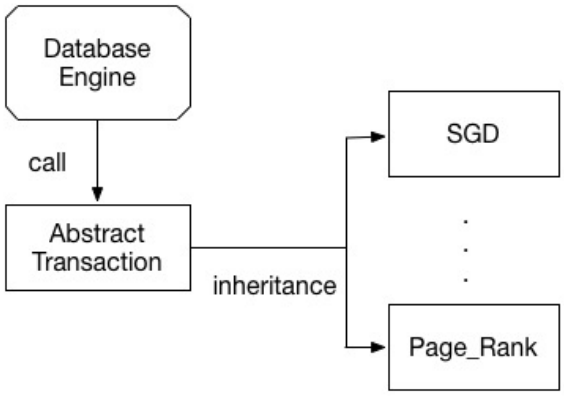


Figure 3: transaction relationship

From this figure, we can see that abstract transaction defines the interface between transaction and database engine. Each transaction has its local state for caching states. The local state usually includes pointer to tables, hyper-parameters of ML algorithms, etc. The transaction interface is composed of following seven functions:

```
private:
    retrieve_state();
public:
    begin(state);
    return_action execute();
    return_action validate();
    int commit();
    int abort();
    int is_converged();
```

The usage of these functions is listed below:

- `retrieve_state()`: cache required data between iterations
- `begin()`: initialize local state for transactions

[2]https://en.wikipedia.org/wiki/Curiously_recurring_template_pattern

- `execute()`: algorithm calculation implementation
- `validate()`: check whether the update is valid; get locks for parameters
- `commit()`: commit transaction. In our case, update parameters
- `abort()`: abort transaction
- `is_converged()`: check whether a transaction need to be rescheduled

As described in section 1.2, these methods would be the programming interface for a transaction to perform a piece of ML computation. The executor will use these methods and reschedule the transactions. All of these as an integration will parallel and complete the algorithm. In the base transaction class, all these functions are prefixed by **super_** to different them from the interface functions.

3 Use Cases

In this section, we will provide two use cases — page rank and support vector machine (SVM) — to show how our transaction mechanism works to compute data intensive algorithm.

3.1 Page Rank

In page rank algorithm[3], each transaction is responsible for one node. We use metis k-way graph partition[4] to divide all these nodes into different groups, which would be a preprocessing step. The corresponding transactions are then assigned to one bulk.

The local states of each transaction and their meanings are listed in table 1.

bool SYNC_FLAG	flag to indicate whether it's synchronous
int VERTEX_NUM	total number of nodes in the graph
vertex_table* nodes	pointer to vertex table
edge_table* edges	pointer to edge table
vector<node_info> incoming_edge	list of incoming nodes
vector<node_info> outgoing	list of outgoing nodes
int id	node id
float cur_pg	read page rank value
float new_pg	calculated page rank value
int pg_version	read page rank version
int node_count	the number of outgoing edges
bool converged	convergence of node

Table 1: local states for page rank transaction

Some of these meta data are queried and filled in the first pass of a transaction, while some of them needed to be retrieve in every pass.

[3]http://www.math.cornell.edu/ mec/Winter2009/RalucaRemus/Lecture3/lecture3.html
[4]https://metis.readthedocs.io/en/latest/

Algorithm 1 is the pseudo code for page rank super function.

Our approach supports both synchronous and asynchronous modes. The main difference is that in asynchronous mode, read and write operations won't return error and in `superCommit` transaction won't check outgoing edges. As a result, there won't be repair in this mode.

3.2 SVM

We use mini-batch stochastic gradient descent (SGD) to train SVM model. We mainly refer to HOGWILD! [1] algorithm. The cost function of sparse SVM is as follows:

$$min_{\omega} \sum_{i} (max(1 - y_i \omega^T x_i, 0) + \lambda \sum_{u \in e_i} \frac{x_u^2}{d_u})$$

Here y_i denotes sample label, x_i denotes value vector of sample. ω is the shared parameter vector. The second part is the regularization part, where e_i denotes the index of components that are non-zero in z_α, and d_u denotes the number of training examples which are non-zero in component u.

The local states of each transaction and their meanings are listed in table 2.

With the help of these local states, sgd_transaction can cache hyper-parameters, as well as useful pointers to database fields.

Algorithm 2 is the pseudo code for each super function. Our code supports both synchronous and asynchronous algorithms. For synchronous mode, every time a transaction validates, it will first lock the parameter component that will be updated. Then transaction will check version distance between the version of parameter component used to calculate and the current version. If it exceeds staleness boundary, this transaction will be aborted. Each time a transaction commits, the current number of iteration will be increased by 1.

Algorithm 1: page rank transaction pseudo code

```
superRetrieve
    retrieve information of pg_version, converged, cur_pg;
    retrieve information of node_count, incoming_edge, outgoing if they are emtpy;
superBegin
    initialize transaction state;
superExecute
    superRetrieve();
    tele_value = 1/VERTEX_NUM if no incoming edges then
        new_pg = tele_value;
    else
        sum = 0;
        foreach incoming node N_i do
            pg = read pg_version page rank value of N_i;
            if read Error then
                add N_i in repair_list;
            sum += 1/node_count_i
        if repair_list then
            return REPAIR
        else
            new_pg = sum * 0.85 + tele_value * 0.15;
    return VALIDATE;
superValidate
    if SYNC_FLAG = False then
        return COMMIT;
    foreach outgoing node N_i do
        if converged then
            continue;
        if current version to far from updating N_i then
            add N_i to repair_list;
    if repair_list then
        return REPAIR
    return COMMIT;
superCommit
    write page_rank value;
    if write Error then
        converged = check converge of node;
        return 1;
    if |new_pg − cur_pg| < 0.001 or no incoming edge then
        set converged
    return 0;
superConverge
    return whether converged;
```

Algorithm 2: SGD transaction pseudo code

```
superBegin
    inputs : initial_state
    initialize transaction state;
superExecute
    randomly choose a subset of sample;
    foreach sample do
        evaluate gradient descent G(x);
        record G(x) in update_map;
    return VALIDATE;
superValidate
    if SYNC_FLAG = True then
        foreach dimension IN update_map do
            lock parameter;
            add locked index to lock_list;
            if version difference > STALENESS then
                unlock all in lock_list;
                return ABORT;
    return COMMIT;
superCommit
    foreach dimension IN update_map do
        update parameter;
        unlock paramter;
    iter_count += 1;
superAbort
superConverge
    if iter_count < iter_total then
        return false;
    else
        return true;
```

bool SYNC_FLAG	flag to indicate whether it's synchronous
int VERTEX_NUM	number of samples to process in each iteration
int STALENESS	version distance allowed to update parameter
sgd_table* sgds	pointer to dataset table
int iter_total	the total number to reschedule the transaction
int iter_count	current reschedule times
double learning_rate	learning step
double lambda	coefficient for regularization
vector<int>* indexs	vector stores sample indexes to train
map<int, double> update_map	map stores parameter index and gradient descent to update
parameter* paramter	pointer to paramter array
parameter** lock_list	list of pointers to parameters that has already been locked

Table 2: local states for sgd transaction

4 Experiment

4.1 Environment

The experiments were conducted on the environment described in table 3.

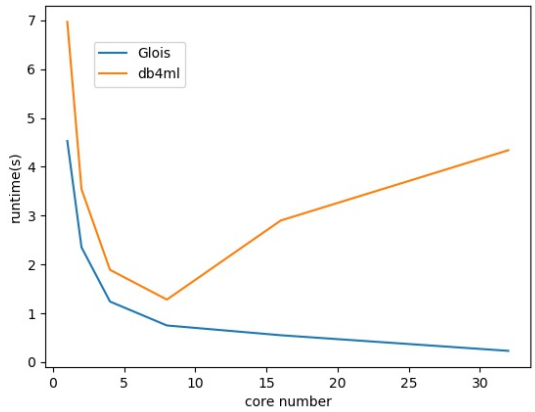
component	**configuration**
kernel version	#57-Ubuntu SMP Tue Jul 15 03:51:08 UTC 2014
kernel release	3.13.0-32-generic
architecture	x86_64
CPU type	Intel(R) Xeon(R) CPU E7- 8830 @ 2.13GHz
CPU number	128
NUMA nodes	8
memory free	455 GB

Table 3: experiment environment configurations

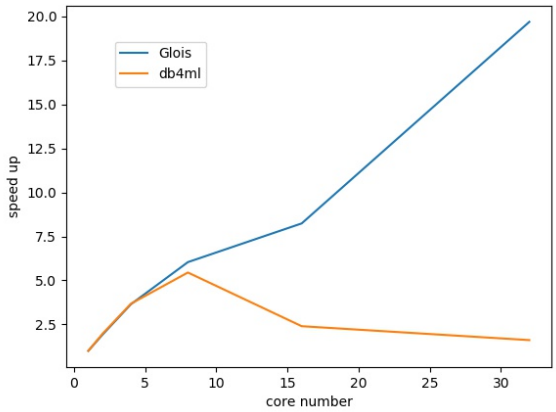
4.2 Page Rank

We used Glois [2] as the baseline of our approach. Figure 4 shows the performance of them. When there were less then 8 cores, Glois and database engine had the same trend and nearly the same performance, though Glois was slightly better. However, when the number of cores increased, database engine performed worse.

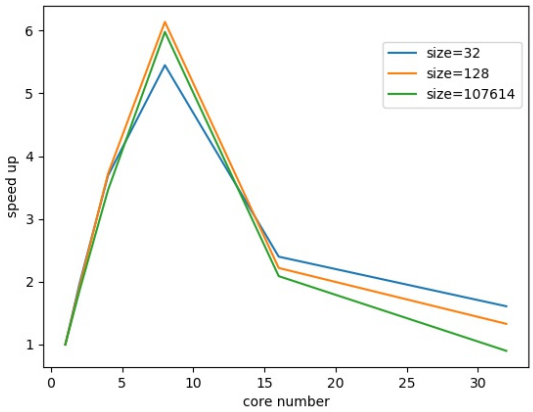
One possible reason why transaction couldn't scale well might be the batch size of 32 group is not the most optimized one. However, according to figure 4(c), different batch sizes actually have the same performance. So this issue needs further exploration for sure.



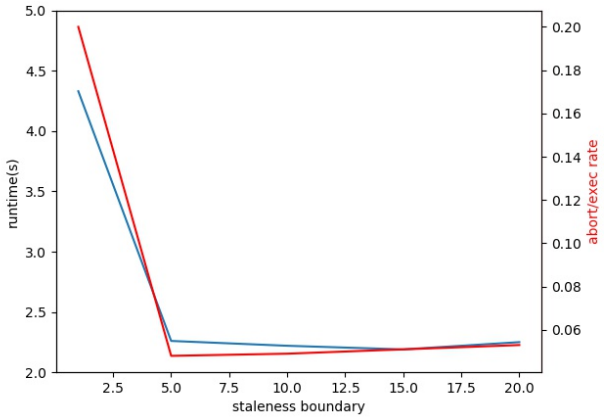
(a) runtime



(b) speedup



(c) batch size



(d) staleness

Figure 4: page rank experiment results

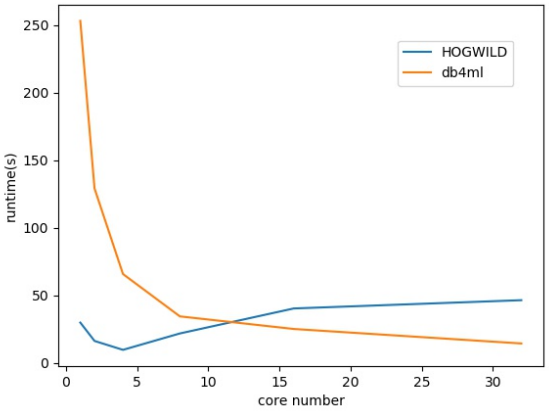
Figure 4(d) shows the performance of synchronous mode algorithm on **8 cores** in terms of staleness. It is clear that abort/exec rate has the same trend with runtime, both of them have a significant decrease when staleness goes to 5 and then they tend to be stable. However, compared with 4(a), the synchronous mode algorithm still perform worse than asynchronous mode. There are two reasons for this:

- Although abort/exec rate is stable, the transaction still has overhead for aborting because of failing in validation step of a too advanced node.
- Even if there is no abort, the system overhead to get a specific version of neighbor's page rank value and does check still exist.

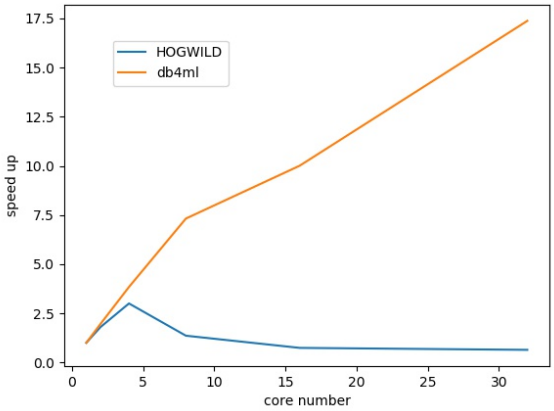
4.3 Sparse SVM

Figure 5 shows the experiment results of SVM-SGD algorithm performance with HOGWILD! and our database system on RCV1 dataset. HOGWILD ran with 0.1 initial step, 1 lambda, while database engine ran with 32 transaction groups in asynchronous mode. Both of them ran for 20 iterations.

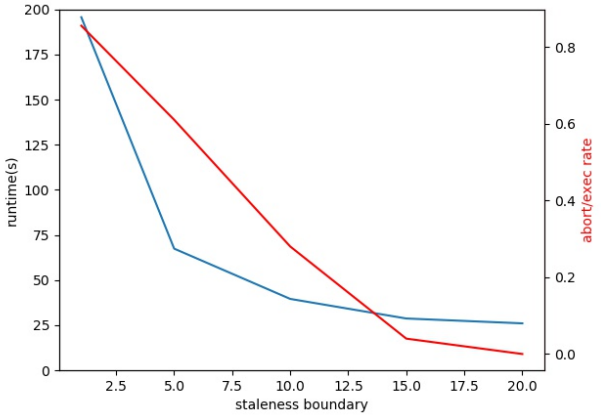
From figure 5(a), we can see that HOGWILD was much faster when core number was small, and it reached the best performance when there were 4 cores. However, while the number of cores continued to increase, the run time of HOGWILD increased systematically. This might looks surprising at the very beginning, but it is inevitable since HOGWILD requires a communication step across all the threads at the end of each epoch. We also observed that when the operating system spread the threads around different NUMA regions, HOGWILD would perform even worse.



(a) runtime



(b) speedup



(c) abort/exec rate

Figure 5: SGD-SVM experiment results

On the other hand, our database engine has a much better speed up compared to HOGWILD according to figure 5(b). When there were 32 cores, our system only ran for 14.7 seconds, resulting

in a speed up of 17.38. We also noticed that both approaches had a stable train RMSE as the number of cores increased — for HOGWILD, it was around 0.33, and for database engine it was 0.52. HOGWILD had a slightly better performance on accuracy.

We further explored the synchronous mode for SVM algorithm in database engine. Figure 5(c) shows how runtime and abort rate changes with staleness boundary on **16 cores**. It is obviously that while the staleness increased, both of them decreased. And when staleness reached 20, the abort rate was 0 and it performed nearly the same with asynchronous mode.

5 Conclusion & Future Work

Using iterative transaction is an innovative approach for computing graph and ML algorithms. This relatively general way that offered by transaction can achieve comparable performance compared to other specific computation engines. However, it also has many disadvantages under some scenarios, as we have shown in section 4.

In the future, we need to explore more attributes of this transaction mechanism and more experiments need to be done. Furthermore, other aspects of transactions should be leveraged in order to improve the performance, like rollback.

References

[1] Recht, Benjamin, et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent." *Advances in neural information processing systems*. 2011.

[2] Pingali, Keshav, et al. "The tao of parallelism in algorithms." *ACM Sigplan Notices*. Vol. 46. No. 6. ACM, 2011.

A DB4ML Instruction[5]

A.1 Build

This system is developed and tested in Ubuntu 16.04. Prior to run it, you should make sure two packages — `libnuma-dev` and `libboost-all-dev` are installed in your system. DB4ML uses CMake to build the whole system, that is `CMakeLists.txt` under project directory. Note that the minimum cmake version you can use is 2.8. To build the system, follow the steps below:

- construct the file `build` under project directory.
- `cd build` file.
- use command `cmake ..` to generate file.
- use command `make` to build the file.

A.2 Run

All the parameters you can use to run the code are listed below:

- `a`: algorithm choice. you can use either `page_rank` or `sgd_svm`.
- `f`: flag of synchronous. Use `true` for synchronous and `false` for asynchronous.
- `s`: staleness boundary.
- `n`: number of cores.
- `g`: number of transaction groups.
- `v`: vertex number (graph) or the number of samples (svm).
- `d`: parameter dimension.
- `i`: iteration number.
- `V`: path to vertex file or sample file.
- `S`: path to src file.
- `D`: path to dest file.
- `l`: learing rate.

`config.h` under `simpledb` directory contains other configurations that are hard coded during compile time. The only parameter you should touch here is `IS_NUMA`. Set it to true when you test the software on bbsn and false when your program runs locally.

Not all parameters are needed for a algorithm. To run page rank, you only need to use `a, f, n, D, S, V, s, r, g, v`, while for sgd_svm they are `a, f, n, v, d, i, s, g, V`. Two examples are shown here:

```
./db4ml -a page_rank -f false -n 8 -D gplus_dest.csv -S gplus_src.csv -V
vertex_32.csv -s 10 -r 1 -g 32 -v 107614
```

[5]This section is pretty much the same with README.md in db4ml repository.

```
./db4ml -a sgd_svm -f true -n 8 -v 23149 -d 47236 -i 5 -s 10 -g 8 -V rcv1_new.txt
```

B Data Format

Currently we use Google+ to test graph algorithm and RCV1 to test SVM. A specific data format is needed for each algorithm. There are three files needed for page rank algorithm: vertex file, src file and dest file.

- `vertex file`: a csv file contains node id, number of output edges, group id.
- `src file`: a csv file contains node id, a list of input node id.
- `dest file`: a csv file contains node id, a list of output node id.

The format for both src file and dest file is:

```
neighbor_number,node_id,neighbor1_id,neighbor2_id, ....
```

Only one file is needed for svm algorithm: a txt file with the format of

```
label  dimension1|value1 dimension2|value2 ... dimensionN|valueN
```

Where label can only be +1 or -1, and values can be float or scientific counting method.

We also provide Python script to preprocess raw data. These scripts are `simpledb/data_creator/graphdataset.py`, `simpledb/data_creator/SGDdataset.py` and `create_data.py`. You can refer to the corresponding script for more details.