Parallel Time Series Modeling - A Case Study of In-Database Big Data Analytics

Hai Qian[1], Shengwen Yang[1], Rahul Iyer[1], Xixuan Feng[1] ⋆, Mark Wellons[2] ⋆⋆, and Caleb Welton[1]

[1] Predictive Analytics Team, Pivotal Inc.
[2] Amazon.Com Inc.

Abstract. MADlib is an open-source library for scalable in-database analytics. In this paper, we present our parallel design of time series analysis and implementation of ARIMA modeling in MADlib's framework. The algorithms for fitting time series models are intrinsically sequential since any calculation for a specific time t depends on the result from the previous time step $t-1$. Our solution parallelizes this computation by splitting the data into n chunks. Since the model fitting involves multiple iterations, we use the results from previous iteration as the initial values for each chunk in the current iteration. Thus the computation for each chunk of data is not dependenton on the results from the previous chunk. We further improve performance by redistributing the original data such that each chunk can be loaded into memory, minimizing communication overhead. Experiments show that our parallel implementation has good speed-up when compared to a sequential version of the algorithm and R's default implementation in the "stats" package.

keywords: parallel computation, time series, database management system, machine learning, big data, ARIMA

1 Introduction

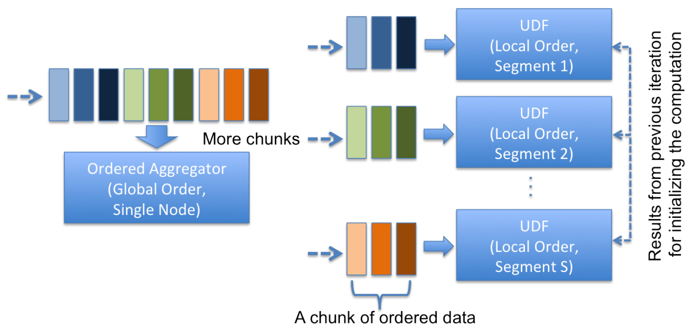
Time series analysis plays an important part in econometrics, finance, weather forecasting, earthquake prediction and many other fields. For example, one of the most conspicuous data analytics task, stock price forecasting, falls into the category of time series analysis. Unlike other data analytics, time series data has a natural temporal ordering. And many time series modeling methods, such as autoregressive integrated moving average (ARIMA) [4] and Cox proportional hazards [7], therefore depend on sequential processing of time series data, which raises a challenge for the data-parallel implementation.

In this paper, we present our parallel implementation of ARIMA, a popular and important time series analysis model, in MADlib [14]. Although all algorithms for ARIMA modeling process the data sequentially, we can still split the

⋆ The first four authors contributed equally to this work.
⋆⋆ Mark Wellons was an intern with Pivotal Inc. during the completion of this work.

data into multiple chunks, each containing consecutive parts of the time series. Each chunk needs the computation result from its neighboring chunk as initial values for its computation. Since the algorithm executes over multiple iterations, we can overcome this limitation by using the results of the preceding chunk from the previous iteration instead of that of the current iteration. The key idea is to take advantage of the iterative nature of the learning algorithms and rely on only local ordering, as illustrated in Fig. 1. At the time of convergence, there is no difference between the values from the previous iteration and the values from the current iteration.

Finally, it is important to note that the data in the database is not necessarily ordered. For fitting models like ARIMA, where the data has to be processed in the fixed order of time, one has to order the data in every iteration, which is time consuming. We avoid this by chunking, sorting, and re-distributing the data accross the segments so that a chunk of ordered data can be read into memory all at once in a segment. This technique not only avoids ordering the data repeatedly but also decreases both the I/O overhead and database function invocation overhead.



Ordered Aggregator (Global Order, Single Node)
More chunks
UDF (Local Order, Segment 1)
UDF (Local Order, Segment 2)
UDF (Local Order, Segment S)
A chunk of ordered data
Results from previous iteration for initializing the computation

Fig. 1: Comparison between sequential and distributed designs. A segment is an independent database process in a shared-nothing distributed MPP (massively parallel processing) database.

The implementation is part of an open-source, scalable, in-database analytics initiative, MADlib [14], maintained by the Predictive Analytics Team at Pivotal Inc [1]. It provides data-parallel implementations of mathematical, statistical and machine-learning methods for structured and unstructured data.

1.1 Related Work

Parallel and Scalable Implementations. The problem of designing parallel algorithms has attracted much attention (see [3] for recent tutorials). Significant efforts have been spent on parallelizing various intricate machine learning algorithms, including k-means++ [2], support vector machines [16] and conditional random fields [5]. This work focuses on parallelizing time series analysis algorithms, where the training process requires a global ordering.

Machine Learning in Databases. Database management systems, being the best in data storage, operation, and analysis for many years, are also studied for data mining and machine learning on large datasets [9, 12]. Ordonez [10] suggested sufficient statistics can be efficiently computed for an important set of models. Feng et al. [8] proposed an architecture linking an essential database programming model, user-defined aggregates, to convex optimization. These techniques, however, did not address time series analysis models in the context of databases.

1.2 What is MADlib?

MADlib is an analytics platform developed by the Predictive Analytics Team at Pivotal Inc. (previously Greenplum). It can be deployed either onto the Greenplum database system (an industry-leading shared-nothing MPP database system) or the open-source PostgreSQL database system. The package itself is open-source and free. The MADlib project was initiated in late 2010 from a research idea by Cohen et. al. [6] who suggested a new trend of big data analytics requiring advanced (mathematical, statistical, machine learning), parallel and scalable in-database functions ("MAD" stands for "magnetic", "agile", and "deep" - see [6] for more details on each property).

By itself MADlib provides a pure SQL interface. To better facilitate data scientists from the R [11] community, there also exists a R front-end package called PivotalR [15]. On Greenplum database (GPDB) systems, MADlib utilizes the data parallel functionality. The calculation is performed in parallel on multiple segments of GPDB, and results from the segments are merged and then summarized on the master node. In many cases, multiple iterations of such calculations are needed. The core functionality for each iteration is implemented in C++ and Python is used to collate results from all iterations.

Although MADlib does not perform parallel computation on the open-source database system PostgreSQL, it is still valuable for processing large data sets that cannot be loaded into memory. For example, in this paper we will describe a comparison between our implementation of ARIMA and the 'arima' function in R's 'stats' package. For building an ARIMA model for a time series data set with the length 10^8, MADlib on PostgreSQL has about the same execution time as R, while consuming only 0.1% of the memory used by R. Thus, MADlib provides the ability of processing large data sets to open-source users for free.

MADlib has various modules including linear, logistic, multinomial logistic regression, elastic-net regularization for linear and logistic regressions, k-means, association rules, cross validation, matrix factorization methods, LDA, SVD and PCA, ARIMA and many other statistical functions. A detailed user documentation is available online at http://doc.madlib.net. For this paper, we focus on our implementation for ARIMA in MADlib.

2 Implementation of ARIMA

In the next few subsections, we describe the algorithm used to solve the problem of maximization of partial log-likelihood to obtain the optimal values of the coefficients of ARIMA. Then, we point out the reason why it is difficult to make the algorithm run in-parallel. This is a common situation in all algorithms that fit ARIMA models. Next, we describe a generic solution to this problem that can be applied to time series problems. Then we describe a simple method to improve the performance of our algorithm. Finally, we discuss various ways to generalize our method to other algorithms.

2.1 The Algorithm

This section follows the design document for ARIMA on the MADlib website [13].

An ARIMA model is an auto-regressive integrated moving average model. An ARIMA model is typically expressed in the form

$$(1 - \phi(B))Y_t = (1 + \theta(B))Z_t, \tag{1}$$

where B is the backshift operator. The time t is from 1 to N.

ARIMA models involve the following variables:

1. The lag difference Y_t, where $Y_t = (1 - B)^d(X_t - \mu)$.
2. The values of the time series X_t.
3. p, q, and d are the parameters of the ARIMA model. d is the differencing order, p is the order of the AR operator, and q is the order of the MA operator.
4. The AR operator $\phi(B)$.
5. The MA operator $\theta(B)$.
6. The mean value μ, which is set to zero when $d > 0$, or can be estimated by the ARIMA algorithm.
7. The error terms Z_t.

The auto regression operator models the prediction for the next observation as some linear combination of the previous observations. More formally, an AR operator of order p is defined as

$$\phi(B)Y_t = \phi_1 Y_{t-1} + \cdots + \phi_p Y_{t-p} \tag{2}$$

The moving average operator is similar, and it models the prediction for the next observation as a linear combination of the errors in the previous prediction errors. More formally, the MA operator of order q is defined as

$$\theta(B)Z_t = \theta_1 Z_{t-1} + \cdots + \theta_q Z_{t-q}. \tag{3}$$

We assume that

$$\Pr(Z_t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-Z_t^2/2\sigma^2}, \quad t > 0 \tag{4}$$

and that $Z_{-q+1} = Z_{-q+2} = \cdots = Z_0 = Z_1 = \cdots = Z_p = 0$.

The likelihood function L for N values of Z_t is then

$$L(\phi, \theta) = \prod_{t=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-Z_t^2/2\sigma^2} \tag{5}$$

so the log likelihood function l is

$$\begin{aligned} l(\phi, \theta) &= \sum_{t=1}^{N} \ln\left(\frac{1}{\sqrt{2\pi\sigma^2}} e^{-Z_t^2/2\sigma^2}\right) \\ &= \sum_{t=1}^{N} -\ln\left(\sqrt{2\pi\sigma^2}\right) - \frac{Z_t^2}{2\sigma^2} \\ &= -\frac{N}{2}\ln\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\sum_{t=1}^{N} Z_t^2 \ . \end{aligned} \tag{6}$$

Thus, finding the maximum likelihood is equivalent to solving the optimization problem (known as the conditional least squares formation)

$$\min_{\theta,\phi} \sum_{t=1}^{N} Z_t^2. \tag{7}$$

The error term Z_t can be computed iteratively as follows:

$$Z_t = Y_t - F_t(\phi, \theta, \mu) \tag{8}$$

where

$$F_t(\phi, \theta, \mu) = \mu + \sum_{i=1}^{p} \phi_i (Y_{t-i} - \mu) + \sum_{i=1}^{q} \theta_i Z_{t-i} \tag{9}$$

Levenberg-Marquardt algorithm (LMA), also known as the damped least-squares (DLS) method, provides a numerical solution to the problem of minimizing a function, generally nonlinear, over the function's parameter space. These minimization problems arise especially in least squares curve fitting and nonlinear programming.

To understand LMA, it helps to know the gradient descent method and the Gauss-Newton method. On many "reasonable" functions, the gradient descent method takes large steps when the current solution is distant from the true solution, but is slow to converge when the current solution is close to the true solution. The Gauss-Newton method is much faster for converging when the current iterate is in the neighborhood of the true solution. The LMA tries to achieve the best of both worlds and combine the gradient descent step with the Gauss-Newton step in a weighted average. For iterates far from the true solution, the step favors the gradient descent step, but as the iterate approaches the true solution, the Gauss-Newton step dominates.

Like various numeric minimization algorithms, LMA is an iterative procedure. To start a minimization, the user has to provide an initial guess for the parameter vector, p, as well as some tuning parameters τ, ϵ_1, ϵ_2, ϵ_3, and $k_{\max}$. Let $Z(p)$ be the vector of calculated errors (Z_t's) for the parameter vector p, and let $J = (J_1, J_2, \ldots, J_N)^T$ be a Jacobian matrix.

A proposed implementation is shown in Algorithm 1.

```
Input: An initial guess for parameters $\boldsymbol{\phi}_0, \boldsymbol{\theta}_0, \mu_0$
Output: The parameters that maximize the likelihood $\boldsymbol{\phi}^*, \boldsymbol{\theta}^*, \mu^*$
$k \leftarrow 0$; $v \leftarrow 2$; $(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) \leftarrow (\boldsymbol{\phi}_0, \boldsymbol{\theta}_0, \mu_0)$;
Calculate $Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$ with Eq. (9). // Vector of errors;
$A \leftarrow J^T J$ // The Gauss-Newton Hessian approximation;
$u \leftarrow \tau * \max_i(A_{ii})$ // Weight of the gradient-descent step;
$g \leftarrow J^T Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$ // The gradient descent step;
stop $\leftarrow (\|g\|_\infty \leq \epsilon_1)$ // Termination Variable;
while not stop and $k < k_{\max}$ do
    $k \leftarrow k + 1$;
    repeat
        $\delta \leftarrow (A + u \times \mathrm{diag}(A))^{-1} g$ // Calculate step direction;
        if $\|\delta\| \leq \epsilon_2 \|(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)\|$ then // Change is too small to continue.
            stop $\leftarrow$ true;
        else
            $(\boldsymbol{\phi}_{new}, \boldsymbol{\theta}_{new}, \mu_{new}) \leftarrow (\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) + \delta$ // A trial step;
            $\rho \leftarrow (\|Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)\|^2 - \|Z(\boldsymbol{\phi}_{new}, \boldsymbol{\theta}_{new}, \mu_{new})\|^2)/(\delta^T(u\delta + g))$ ;
            if $\rho > 0$ then // Trial step was good
                $(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu) \leftarrow (\boldsymbol{\phi}_{new}, \boldsymbol{\theta}_{new}, \mu_{new})$ // Update variables;
                Calculate $Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$ with Eq. (9); $A \leftarrow J^T J$; $g \leftarrow J^T Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$;
                stop $\leftarrow (\|g\|_\infty \leq \epsilon_1)$ or $(\|Z(\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)^2\| \leq \epsilon_3)$ ;
                $v \leftarrow 2$; $u \rightarrow u * \max(1/3, 1 - (2\rho - 1)^3)$;
            else// Trial step was bad
                $v \leftarrow 2v$; $u \leftarrow uv$;
            end
        end
    until stop or $\rho > 0$;
end
$(\boldsymbol{\phi}^*, \boldsymbol{\theta}^*, \mu^*) \leftarrow (\boldsymbol{\phi}, \boldsymbol{\theta}, \mu)$;
```

Algorithm 1: A proposed LMA implementation for fitting ARIMA model

Suggested values for the tuning parameters are $\epsilon_1 = \epsilon_2 = \epsilon_3 = 10^{-15}, \tau = 10^{-3}$ and $k_{\max} = 100$.

The Jacobian matrix $J = (J_1, J_2, \ldots, J_N)^T$ requires the partial derivatives, which are

$$J_t = (J_{t,\phi_1}, \ldots, J_{t,\phi_p}, J_{t,\theta_1}, \ldots, J_{t,\theta_q}, J_{t,\mu})^T \tag{10}$$

Here the last term is present only when we want to estimate the mean value of the time series too. The iteration relations for J are

$$J_{t,\phi_i} = -\frac{\partial Z_t}{\partial \phi_i} = Y_{t-i} - \mu + \sum_{j=1}^{q} \theta_j \frac{\partial Z_{t-j}}{\partial \phi_i} = Y_{t-i} - \mu - \sum_{j=1}^{q} \theta_j J_{t-j,\phi_i}, \quad (11)$$

$$J_{t,\theta_i} = -\frac{\partial Z_t}{\partial \theta_i} = Z_{t-i} + \sum_{j=1}^{q} \theta_j \frac{\partial Z_{t-j}}{\partial \theta_i} = Z_{t-i} - \sum_{j=1}^{q} \theta_j J_{t-j,\theta_i}, \quad (12)$$

$$J_{t,\mu} = -\frac{\partial Z_t}{\partial \mu} = 1 - \sum_{j=1}^{p} \phi_j - \sum_{j=1}^{q} \theta_j \frac{\partial Z_{t-j}}{\partial \mu} = 1 - \sum_{j=1}^{p} \phi_j - \sum_{j=1}^{q} \theta_j J_{t-j,\mu}. \quad (13)$$

Note that the mean value μ is considered separately in the above formulations. If we do not want to estimate the mean value, μ will be simply set to 0. Otherwise, μ will also be estimated together with $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$. The initial conditions for the above equations are

$$J_{t,\phi_i} = J_{t,\theta_j} = J_{t,\mu} = 0 \quad \text{for } t \leq p, \text{ and } i = 1, \ldots, p; j = 1, \ldots, q\ , \quad (14)$$

because we have fixed Z_t for $t \leq p$ to be a constant 0 in the initial condition. Note that J is zero not only for $t \leq 0$ but also for $t \leq p$.

2.2 Problems in Parallelization

It is easy to see that Eqs. (8, 11, 12, 13) are difficult to parallelize. Each step computation uses the result from the previous step. Therefore, we have to scan through the data sequentially to compute the quatities in these equations, and we have to do this in every iteration. If the data set is large then it is time-consuming to use this algorithm.

2.3 Our Solution

In order to utilize the data parallel capability of MPP(massively parallel processing) database, we propose to split the whole time series data into a set of sorted chunks numbered from 1 to N, each containing a sequence of consecutive time series data of size K. During the same time, we re-distribute the chunks of data onto all segments of the MPP database so that we could process them in parallel.

Since the model fitting involves multiple iterations of computations, for any subset i except for the first one, we use the results of the $(i-1)$-th subset from the previous iteration as the initial values. The first subset's initial values are known to be 0. Thus in each iteration, the computation for each subset of data does not need to wait for the results from the previous subset. In this way, the model fitting computation can be done in parallel.

Further, we find that aggregating each subset of consecutive time series data into an array (i.e. a data chunk) can greatly simplify the implementation (for example, from a stateful window function implementation to a plain UDF implementation) and accelerate the computation (mainly due to the reduction of I/O overhead and function call overhead). Furthermore, the value of size K is chosen in such a way that each chunk of data can be completely loaded into the available memory.

The following SQL script shows how we split and redistribute the data according to the above proposals.

```
-- redistribute the consecutive time series data into a same segment
create temp table dist_table as
select
  ((tid - 1) / chunk_size)::integer + 1 as distid, tid, tval
from input_table
distributed by (distid)

-- insert the preceding p data points for each chunk
insert into dist_table
select o.distid + 1, tid, NULL, tval
from (
  select distid, max(tid) maxid
  from dist_table
  where distid <> (select max(distid) from dist_table)
  group by distid
) q1, dist_table o
where q1.distid = o.distid and q1.maxid - o.tid < p

-- aggregate each chunk into an array to avoid repeated ordering
-- and communication overhead
create temp table work_table as
select
  distid, (distid - 1) * chunk_size + 1 as tid,
  array_agg(tval order by tid) as tvals
from dist_table
group by distid
distributed by (distid)
```

Our experiments show that the convergence of our implementation is very good. The 'chunk size' parameter does not have a significant impact on the performance as long as it is not too small, which can make the chunking meaningless, or not too large, which can make the data re-distribution difficult. As is shown in Fig. (2), the execution times are around $400 \sim 500$ secs for different chunk sizes. The stable execution time for different chunk sizes makes it easier for the user to choose the proper parameters for the algorithms.

3 Experimentation

In this section, we present a set of experiments to measure the performance of our parallel implementation of ARIMA.

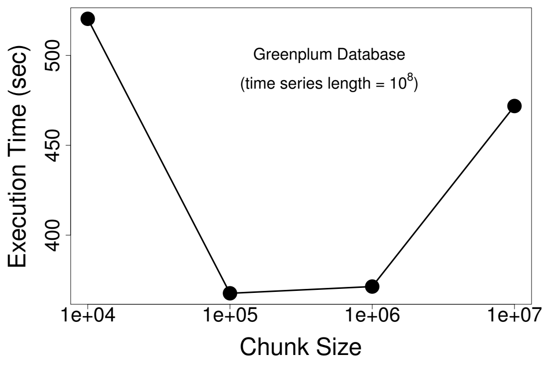


Fig. 2: We fit a time series with length of 10^8 using different chunk sizes. The execution times are around $400 \sim 500$ secs, and are quite stable.

Configuration. We did our experiments on a DCA (Data Computation Appliance) produced by EMC Corporation, containing a Greenplum database system installed with 48 segments. The data sets that we used for the experiments are generated by R's "arima.sim". We generated multiple time series with different lengths. The data set with the length 10^9 is too large to be generated by R directly. Instead we first generated 10 pieces of time series with the lengths 10^8, and then assemble them together to form the complete time series.

3.1 Scalability

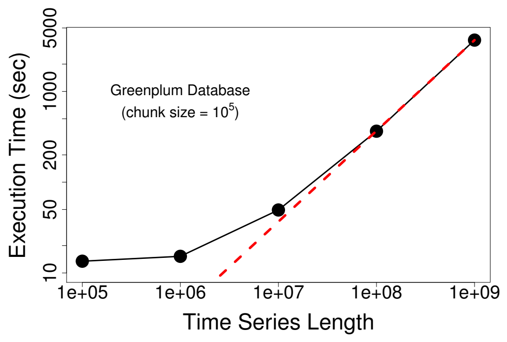
First, we measure the execution time of our implementation applied onto time series with different lengths. We run the tests in both Greenplum database and PostgreSQL database.

As is shown in Fig. 3, the execution time for large data sets is almost linear with respect to the length of the time series. For smaller data sets, the communication overhead between the multiple segments has a negative impact and the execution time is larger than the time for a pure linear execution time.

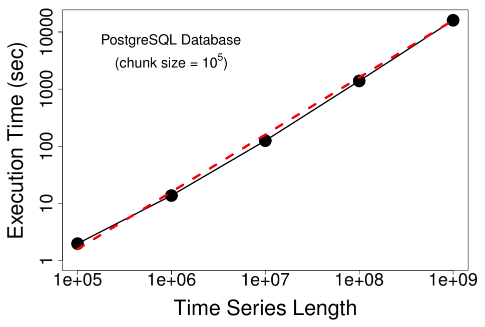
PostgreSQL database does not have the overhead of merging results from multiple segments, and thus the execution time, as shown in Fig. 3, is a linear function of the data size.

3.2 Total runtime comparisons

In Table 1, we compare the execution times of ARIMA model fitting in R and in MADlib on Greenplum database and PostgreSQL database. The execution time of MADlib's ARIMA on PostgreSQL is approximately the same as R (actually it is a little faster), but the memory usage is only a tiny fraction of R's "arima" function. This is because R loads all data into memory for processing, while the database systems esentially load one row of data into memory for processing and then proceed to the next row.



(a)



(b)

Fig. 3: (left) We fit time series with MADlib's ARIMA function on GPDB (left) and PostgreSQL(right). The chunk size in both cases is 10^5. The red dashed line is the fit to $t = \alpha l$, where t is the execution time, l is the length of the time series and α is a constant.

Although GPDB uses 48 segments, the speedup over PostgreSQL is about 3X to 4X. This is due to the communication overhead of communicating between multiple segments, especially the part where the data is loaded and re-distributed for sorting. If we compare the time taken for actual computation, GPDB is on average 17.6X faster than PostgreSQL.

	MADlib on GPDB	MADlib on Postgres	R's arima function
Execution Time (sec)	364.4	1391.9	1964.4
Iteration Number	29	29	N/A

Table 1: Here we compare the execution times of ARIMA model fitting in R and in MADlib on Greenplum Database and PostgreSQL database. The time series used to fit the ARIMA model has a length of 10^8. Note that running MADlib's ARIMA on Postgres is not only faster but also uses much less memory (0.1%). Running this data set in R uses almost 70% of the machine's memory (50G memory). The iteration number for R is not available, because R's ARIMA does not output how many times it has iterated.

3.3 Sensitivity of number of segments

For MADlib's ARIMA running on Greenplum database system, we also measured the execution time vs. the number of segments used, which is shown in Fig. 4. Here, we use a time series with the length equal to 10^8. When the number of segments is less than 32, then the execution time decreases as more segments are added. However, increasing the number of segments beyond 32 will increase the execution time due to the increasing communication overhead between segments.

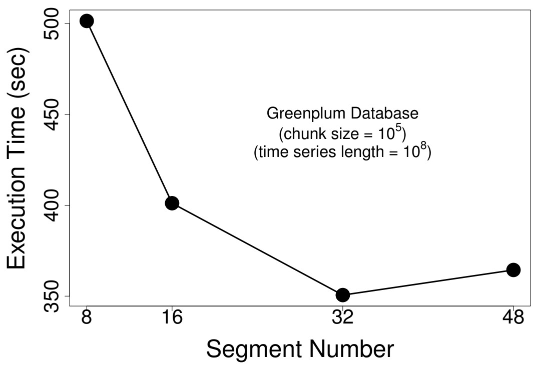


Fig. 4: We fit a time series with length of 10^8 using different numbers of segments in Greenplum database system.

4 Discussion and Conclusion

The notable methods that we used in the implementation of ARIMA in MADlib are: (1) split the data into chunks of consecutive data points and let each chunk use the result of the previous chunk from the previous iteration to initialize the calculation; (2) aggregate the chunk of data points into an array and redistribute the aggregated chunks accross segments so that each chunk of data can be loaded into memory and processed in one single function call by a segment. The first method makes it possible to parallelize the algorithm, and the second method greatly simplifies the implementation and improves the performance.

In this paper, we described our parallel implementation of ARIMA in MADlib and showed significant runtime improvements compared to serial implementations. It is easy to see that the above two methods can be easily generalized to other algorithms. The first method can be applied for any algorithm that requires a global ordering of the data. The second method can be used for improving the performance of any parallel algorithm. We aim to extend MADlib by generalizing this solution to parallelize other time series analysis algorithms.

Bibliography

[1] Pivotal. http://gopivotal.com/, 2013.

[2] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[3] R. Bekkerman, M. Bilenko, and J. Langford. Scaling up machine learning: parallel and distributed approaches. In *Proceedings, 17th ACM SIGKDD Tutorials*, KDD '11 Tutorials, pages 4:1–4:1, 2011.

[4] G. E. Box and D. A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65(332):1509–1526, 1970.

[5] F. Chen, X. Feng, C. Ré, and M. Wang. Optimizing statistical information extraction programs over evolving text. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 870–881. IEEE, 2012.

[6] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. MAD skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2-2:1481–1492, 2009.

[7] D. Cox. Regression models and life-tables. *Journal of the Royal Statistical Society. Series B (Methodological)*, 34(2):187–220, 1972.

[8] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336. ACM, 2012.

[9] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib analytics library: or MAD skills, the SQL. *Proceedings of the VLDB*, 5(12):1700–1711, 2012.

[10] C. Ordonez. Building statistical models and scoring with udfs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. ACM, 2007.

[11] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2013.

[12] S. Sarawagi, S. Thomas, and R. Agrawal. *Integrating association rule mining with relational database systems: Alternatives and implications*, volume 27. ACM, 1998.

[13] The Predictive Analytics Team at Pivotal Inc. Design document for MADlib. http://madlib.net/design.pdf, 2013.

[14] The Predictive Analytics Team at Pivotal Inc. MADlib: An in-database analytics platform. http://madlib.net, 2013.

[15] The Predictive Analytics Team at Pivotal Inc. PivotalR: An R front-end to both GPDB/Postgres and MADlib. http://cran.r-project.org/web/packages/PivotalR/, 2013.

[16] Z. A. Zhu, W. Chen, G. Wang, C. Zhu, and Z. Chen. P-packsvm: Parallel primal gradient descent kernel svm. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 677–686. IEEE, 2009.