Benchmark Big Data Systems on Complex Analytic Queries

Shumo Chu, Edward Xuejun Wu, Xiaoyi Zhang
CSE, University of Washington
Seattle, Washington, USA
{chushumo, edwardwu, xiaoyiz}@cs.uw.edu

ABSTRACT

In recent years, considerable number of big data systems emerged for large-scale data analysis following MapReduce. Most of the systems can handle the big volume of data if the analytic query itself is not complex. In this paper, we would like to study how these systems perform against complex queries. We consider the query is complex if the query has the following properties: 1) the query is iterative. 2) the query contains aggregates or UDFs which are widely used for data analytics. 3) the query processes significant amount of data. We proposed a benchmark consisting of a collection of such queries and evaluated these queries on state-of-art representative big data systems in their categories. We think this empirical evaluation and analysis could be beneficial to the development of next generation big data systems.

1. INTRODUCTION

Following MapReduce, many big data analytics systems emerges in recent years, including Spark-SparkSQL [11, 12], GraphLab [6], Myria [7] and others [1, 9]. One similarity among these systems is that they all deploy in shared-nothing cluster and can scale well for relatively simple queries over large data due to massive parallelism.

In this paper, we ask a vital question, how do these system perform if the query is "complex". We characterize the query as complex if the query has the following properties.

1. **Iterative**. This means that the query requires some iterative computation. Example can be like PageRank and graph reachability.

2. **Aggregation and Filtering**. Aggregation means the final result of the query is aggregated and contains possibly much less data than input. This can be think of aggregation in SQL or `reduce` on data by applying combining function. Data filtering means the input data needs to be filtered by certain predicate of conditions. Both aggregation and filtering is not abnormal in data curating or ETL process.

3. **Multiple data sources**. This requires the input data of the the query is from more than one data source or tables. This requirement is not rare in practice and will require the system to handle data communication properly since in many cases the system need to send data between servers.

We choose this criteria from two different aspects of considerations. On one hand, these properties are not abnormal in analytic processes. For example, to get value from data, many algorithms like PageRank, K-Means require iterative query. Also data analysts usually will spend a large portion of time to do ETL or data curating, which requires the ability of aggregation and filtering. And in real world applications, data can usually comes from different data sources or tables and need to be combined together. On the other hand, these properties requires clever system design and implementation to be efficiently computed. And simple parallelization may not need to satisfactory performance. For example, pipelining hadoop jobs to execute iterative queries will suffer from Hadoop's huge cost of serialization and deserialization and the cost of synchronization between jobs. From these two perspective, we pick these properties to make this benchmark practical yet could be helpful to design future big data systems.

We chooses three queries, least common ancestor in a citation graph, k-core computation and myMergerTree edge computation, which have the properties that we defined, to test the performance of the three big data systems. Our objective evaluation contains two measures, lines of code (LOC), which measures how easy to write code using the programming abstraction offered by a system, and query runtime, which measures the efficiency of the system.

We test our benchmark in three state of art big data systems. Myria, a shared nothing parallel relational database system, whose programming model is SQL with iteration. Spark, a data flow based distributed computation engine, whose programming model is MapReduce. GraphLab, a distributed graph processing and machine learning engine, whose computation model is vertex based message passing and aggregation (mainly support Gather, Apply and Scatter operations).

Our empirical evaluation shows that Myria has the fewest lines of code in all three queries and has the fastest runtime in two of three queries (with the exception of LCA, which is 1.38x of GraphLab's runtime). Spark is comparable with Myria in LOC but is slowest among the three systems among the two queries on which all three system successfully completed. GraphLab shows very good effciency however needs the most lines of code among all systems.

We also discussed our experience of using the three system in Section 4.2.

2. METHODOLOGY

2.1 Systems that we compare

We choose 3 different big data systems in different categories.

Myria. Myria [7, 10] is a new shared nothing parallel relational database management system (RDBMS) with focus on complex workflows from science. It uses relational data model and embraces many modern technologies and architectural decisions which are widely used in big data systems. We choose Myria as the representative of state of art distributed relational big data systems.

Spark. Spark [12] is a distributed computing engine with significant performance improve over Hadoop MapReduce. It has an DAG execution engine that support cyclic data flow and in memory computing. We choose Spark as the representative of state of art MapReduce like system.

GraphLab. GraphLab [6] is a distributed graph computation engine for real world large scale graphs. We choose GraphLab as the representative of state of art graph processing system.

2.2 Metrics

The advantages brought by big data system are the ability of process large amount of data using high level abstraction. So we measure two metrics:

Lines of code (LOC). One advantage of modern big data systems is to allow users to write high level code without worrying details of data communication. Thus, we use lines of code as a measure of degree of abstraction of the DSL that these system uses. For fairness, we define the following rules:

1. Calling application specific libraries is not allowed.
2. Each line of the program should not exceed 80 characters.
3. Comment does not count into LOC.

Runtime. We deployed three big data system in Amazon EC2. For each system, we use the same number (1 masters, 16 workers) and the same type (m3.large) of EC2 instances. The runtime does not include data loading time.

3. BENCHMARK QUERIES

3.1 Q1: LCA

This query is to compute the least common ancestor (LCA) of two academic papers. An ancestor of paper p is transitively defined as:

1. Any paper that p cites is an ancestor of p
2. Any paper that p's ancestor cites is an ancestor of p

Given two paper p_1 and p_2, if their sets of ancestors are $a(p_1)$ and $a(p_2)$, their common ancestors are $a(p_1) \cap a(p_2)$. We define the distance from a common ancestor as $d(a_1, p_1, p_2) = max(dist(a_1, p_1), dist(a_1, p_2))$, where $dist(a_1, p_1)$ is the minimum number of citation hops from a_1 to p_1. Then we can define a total ordering among the common ancestors of p_1 and p_2 as, if $a_i, a_j \in a(p_1) \cap a(p_2)$, $a_i \prec a_j$ if and only if.

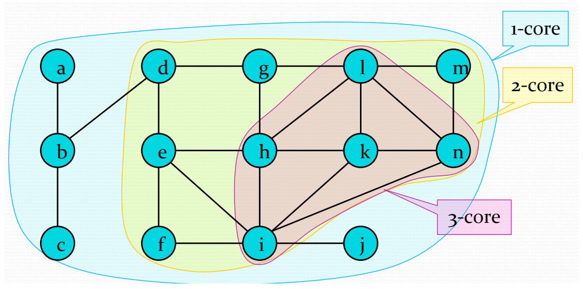


Figure 1: Example of k-cores of a graph G

1. $d(a_i, p_1, p_2) < d(a_j, p_1, p_2)$.
2. $d(a_i, p_1, p_2) = d(a_j, p_1, p_2)$ and the year when a_i published $year(a_i)$ is earlier than a_j's $year(a_j)$.
3. if $d(a_i, p_1, p_2) = d(a_j, p_1, p_2)$ and $year(a_i) = year(a_j)$, a_i's paper id is a smaller number than a_j's.

3.2 Q2: K-Core

The query is to compute the k-core (or k-degenerate graph) of an undirected graph. It is firstly defined by Paul Erdős and Hajnal as color number [5]. k-core is an important structural property and has been studied extensively in network analytics [2, 4].

A k-core of a graph G is a maximal induced subgraph of G in which all vertices have degree at least k in the subgraph. Figure 1 shows 1 . . . 3-cores of a graph.

k-core has two interesting properties. First, as showed in Figure 1, n-core contains all the vertices in $n + 1$-core. Second, there is a polynomial time ($O(m)$, m is the number of edges of the graph) algorithm for core decomposition (compute all cores) [3]. Algorithm 1 shows the algorithm of computing k-core.

Algorithm 1 Core Decomposition Algorithm

```
Input: k                                          ▷ k
Input: G                                          ▷ Input Graph
 1: while true do
 2:     $count \leftarrow 0$
 3:     for all every vertex $v \in V(G)$ do
 4:         if $deg(v) < k$ then
 5:             remove $v$ from $V(G)$
 6:             remove edges adjacent to $v$ from $E(G)$
 7:             $count \leftarrow count + 1$
 8:         end if
 9:     end for
10:     if $count = 0$ then
11:         break.
12:     end if
13: end while
```

3.3 Q3: Merger-Tree

The merger tree query is from large-scale cosmological simulation in astronomy. The astronomers want to track the evolution of galaxies from the Big Bang to the present day, which spans over 14 billion years. The merger tree query compute the hierarchical assembly of galaxies by tracking the merging of small galaxies. In our evaluation, we assume that all the preprocessing has been properly done and evaluate the 3rd computation step in [8]. The query will

```
particle(pid, grp_id, time) :-
   poi(pid, grp_id, time),
   halo(grp_id, time, totalParticles>256).

edges(time, gid1, gid2, $count(*)) :-
   particle(pid, gid1, time),
   particle(pid, gid2, time+1).

treeEdges(1, gid1, gid2, count) :-
   edges(time=1, gid1, gid2, count).

treeEdges(time+1, gid2, gid3, count) :-
   treeEdges(time, gid1, gid2, count),
   edges(time+1, gid2, gid3, count).
```

Figure 2: Merger Tree Query

compute weighted edges (number of particles shared) between two galactic groups in two adjacent time stamp (t and $t + 1$). Also, this query only compute the edges related to a set of particles that specified by the astronomers (Particles of interest).

We show the datalog version of this query in Figure 2. The query firstly uses halo table to filter out all the particles that are in the halo with total number of particles less or equal than 256 (those are considered as insignificant halo by the astronomers). Then we computed all the edges by joining the particle particle table. At last we compute all the tree edges, which only consider the edges that can be traced “back” to current time.

4. EXPERIMENTAL EVALUATION

We report the experimental result of running three benchmark queries from Section 3. We deployed the three systems in Amazon EC2 using the identical configuration, which uses 1 master node and 16 slave nodes. Each node is a m3.large instance which has 2 vCPU, 7.5 GB RAM and 32GB SSD storage. We used the most up to date version of the 3 systems that we can get: Myria (Daily built in Mar 10, 2015, commit: 90d85cd), Spark (1.2.1 release) and GraphLab (Open sourced version in GitHub, commit: 18c2103). All our source code is open sourced in `https://github.com/stechu/CSE544_Project`.

4.1 Objective Evaluation

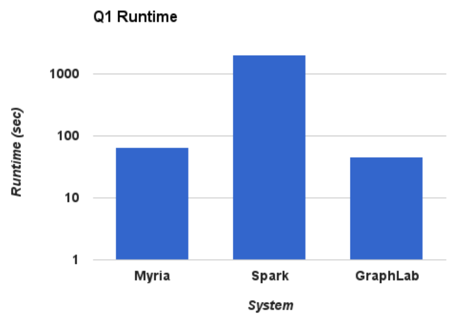
4.1.1 Q1

We evaluate the performance of three systems on Q1 using jstor scientific digital library data. We use two tables:

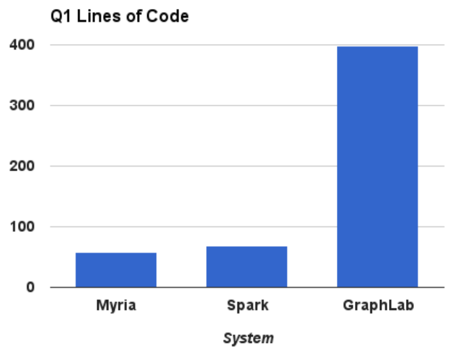
1. Paper(pid:int, year:int) is a table with two columns. The first column is the unique paper id of each paper. The second column is the year when the paper was published. This table contains 1.8 million papers.

2. Citation(p1:int, p2:int) is a table with two columns. Each row of this paper represents a citation. The first column of this table is the citing paper, and the second is the cited paper. This table contains 8.2 million citations.

Figure 3 shows the runtime and lines of code (LOC) using three systems. We can observe the GraphLab has the fastest runtime



(a) Runtime



(b) Lines of Code

Figure 3: Q1: LCA

among three systems. Myria is slightly slower (1.38x runtime) than GraphLab. Spark is slowest among the three systems. Its runtime is 43x GraphLab’s runtime. In term of LOC (Figure 3b), both Myria and Spark use less than 70 lines to express this query while GraphLab need nearly 400 lines. This is due to the programming model and the language support. GraphLab uses C++ and force the programmer to “think like a vertex”.

4.1.2 Q2

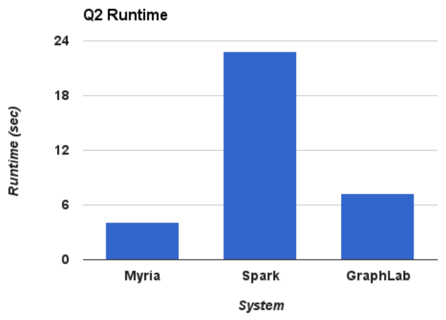
We evaluate the performance of the three system on Q2 using sampled twitter social network data. The table contains two columns, the first column is the id of follower and the second is the id of the followee. Since k-core requires a undirected graph, we add an reversed edge between two nodes if there is a single directed edge between them. The tables contains about 2 million rows.

Figure 4 shows the runtime and lines of code using three systems. We can observe that Myria has smallest runtime among all the three systems. GraphLab has about $1.8x$ Myria’s runtime and Spark has about $5x$ Myria’s runtime. In terms of lines of code, Myria needs only 13 lines of code, while Spark needs 23 lines of code and GraphLab used 146 lines of code.

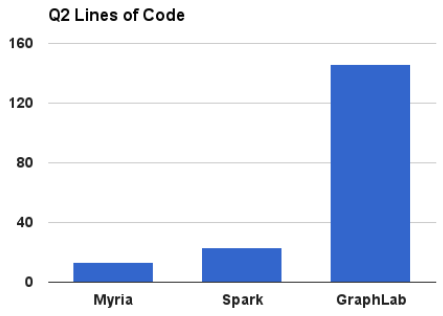
4.1.3 Q3

We evaluate the performance of the three systems on Q3 using 13 snapshots of galaxy data from UW astronomy department. We use two tables:

1. Halo(grpId:long, timeStep:long, mass:double, numParticles: long) is a table with 4 columns. The first two columns form a unique identification (primary key) for a halo. The third



(a) Runtime



(b) Lines of Code

Figure 4: Q2: k-core

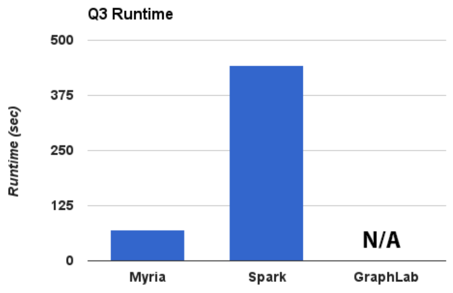
column is the mass of a halo. The forth column contains the number of particles within a halo. This table contains 68K halos.

2. ParticleOfInterest(pid:long, mass:double, type:string, grpId:long, timeStep: long) is a table with 5 columns. The first column is the unique identifier (primary key) of a particle. The second column is the mass of a particle. The third column contains the type of a particle. The last two columns decides which group (halo) a particle belongs to. This table contains 42 million particles.

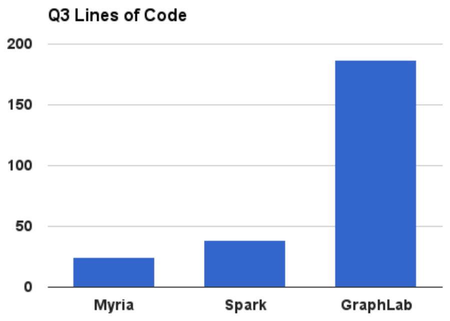
Figure 5 shows the runtime and lines of code using three systems. Compared to the previous 2 queries, query 3 contains a few join operations that Myria and Spark supports really well, but Graphlab doesn't have native support of. In terms of runtime performance, similar to the previous 2 queries, Myria completed first, and Spark was orders of magnitude slower. Graphlab was not evaluated in the runtime performance due to relative difficulty of implementing inner join within its GAS programming abstraction. We later performed a separate evaluation where Myria in pitched with Graphlab for the groupby and count operations. Graphlab was 2x slower probably due to its run including some input data ingestion time, while Myria's run has already preloaded all the data into the cluster. For lines of code, continuing the trend, Myria needed only 25 lines of code, while Spark needs 39 lines of code and GraphLab used 187 lines of code.

4.2 Comparing the three systems

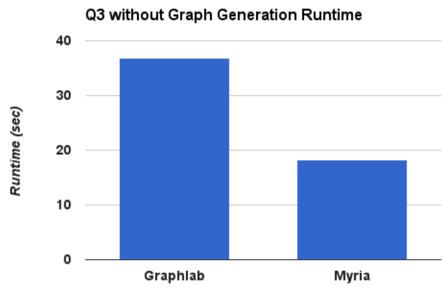
We try to draw some subjective conclusions for the three system from the objective evaluation in Section 4.1 and from our experience of using these three systems in this section.



(a) Runtime



(b) Lines of Code



(c) Runtime without self-join

Figure 5: Q3: merger tree

4.2.1 *Myria*

Myria topped two queries in runtime and topped all three query in LOC, which clearly shows that on the scenario that we defined. Myria is fast and at a very high level of abstraction to make the users who writes analytic queries easier. We think this good performance comes from two reasons:

1. The relational abstraction fits well with "Big Data". The relational data model provide user a declarative language that easy to express analytic workload. On the other hand, extensive research on query optimization will enable efficient parallel evaluation of the relational query.

2. As a state of art distributed shared nothing database system, Myria adopted many new technology such as light weight iterative processing and parallel data flow based back-end engine. The good system implementation contributes to the performance as well.

The inconvenience of using Myria is its data ingestion process. Although the system support ingesting data from S3, the user have to use post a json formatted query to do the data ingestion while cannot direct write S3 bucket as the data source in the query.

4.2.2 *Spark*

Spark is a new MapReduce implementation with significant improvement on iterative and in-memory computing ability. We find that the most significant strengths of spark is its Well matured and system and eco-system. Spark is very easy to deploy in EC2. The programming guide and documentation is very user friendly.

In terms of abstraction level, Spark is below Myria. In Spark python, user still need to use MapReduce like transform function to operate data. It is slightly less convenient compared with Myria. For example, to do a join, the the user need firstly apply a map with a function which transform the joined columns to key. But overall, using Spark python to express all the three query is not hard and uses relatively small LOCs.

The biggest problem of Spark is performance. Spark is the slowest system in 2 of 3 queries. We do not have a concrete idea on the reason for that. But our guess could be the synchronization barrier over iteration in spark is still larger than other modern systems (Myria and GraphLab), although improved largely from Hadoop MapReduce.

Another problem of Spark is lacking automatic control of parallelism of RDD. The parallelism of RDD will be the sum of the two source RDDs if user joins them. This causes the parallelism of RDD grows exponentially if there are joins inside an iteration. This will quickly lead to run out of system sources since the overhead of each RDD container is not negligible. This problem can be solved by forcing keeping parallelism after join. But it is not very easy for a user to identify this problem and find the solution.

4.2.3 *GraphLab*

Graphlab is a high performance, distributed graph based computing framework. Compared to the other big data frameworks, graphlab differs by exploiting the locality and parallelism of most graph analytics algorithms, such as PageRank and k-core. Graphlab allows users to write functional programs that will asynchronously execute on all the vertices. They term their programming paradigm GAS (gather, apply and scatter). Gather functions are executed on all incoming edges, which serves to gather information from neighboring vertices. Apply functions merge the information collected by the gather functions and update the data in the vertex. And finally the scatter functions will scatter the new data in the vertex to its neighbors.

Graphlab has good performance due to being implemented in C++ and leverages all the locality and parallelism that exist in most graph analytics algorithms. In some way, they have offer the similar benefits of a traditional data flow machine, where vertex program is asynchronously executed only when a neighboring vertex has gotten some new information.

However, there are 2 big problems with Graphlab. The first being the restrictiveness of its programming interface. While it is very intuitive to express graph analytic algorithms in Graphlab's GAS model, other types of complex data operations such as joining, filtering are not very straight forward to implement. Secondly, Graphlab is by far the more complex tool to use due to its complicated C++ interface, and the need to write a bunch of housekeeping code. As shown in previous sections, GraphLab requires an order of magnitude more lines of code to implement the same algorithm compared to Spark and Myria. More line of code also means the programmers are going to have a steeper learning curve and more places to make mistakes. Finally, Graphlab is also quite limited in its data ingestion modules, as it only supports loading data from local disk and HDFS. And the user has to manually partition in the input files in order for the file to be loaded in parallel.

5. CONCLUSION

We defined a benchmark which considering three important properties of modern complex analytic queries. We evaluated three state of art big data systems which have different data model and programming abstraction over our benchmark using the same instance setting in EC2. We discuss the pros and cons of these systems and reported our experience on deploying and using these three systems.

6. ACKNOWLEDGMENTS

We would like to thank Daniel Halperin for his MyriaL code on LCA and thank Maaz Ahmad and Antoine Kaufmann for their GraphLab code on LCA. We would like to thank Jennifer Ortiz and Laurel Orr for their help on myMergerTree query. We would like to thank Magdalena Balazinska for the helpful discussion on the project.

7. REFERENCES

[1] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009.

[2] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Advances in Neural Information Processing Systems 18 [Neural Information Processing Systems, NIPS 2005, December 5-8, 2005, Vancouver, British Columbia, Canada]*, pages 41–50, 2005.

[3] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *CoRR*, cs.DS/0310049, 2003.

[4] J. Cheng, Y. Ke, S. Chu, and M. T. Özsu. Efficient core decomposition in massive networks. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 51–62, 2011.

[5] P. Erdős and A. Hajnal. On chromatic number of graphs and set-systems. *Acta Mathematica Academiae Scientiarum Hungarica*, 17(1-2):61–99, 1966.

[6] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 17–30, 2012.

[7] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the myria big data management service. In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 881–884, 2014.

[8] S. Loebman, J. Ortiz, L. L. Choo, L. Orr, L. Anderson, D. Halperin, M. Balazinska, T. Quinn, and F. Governato. Big-data management use-case: A cloud service for creating and analyzing galactic merger trees. In *Proceedings of the Third Workshop on Data analytics in the Cloud, DanaC 2014, June 22, 2014, Snowbird, Utah, USA, In conjunction with ACM SIGMOD/PODS Conference*, pages 1–4, 2014.

[9] 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 *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005, 2010.

[10] UWDB. Myria. `http://myria.cs.washington.edu/`, 2014.

[11] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In K. A. Ross, D. Srivastava, and D. Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 13–24. ACM, 2013.

[12] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.