UDC 004.75

M.V. TALAKH, Yu.O. USHENKO, E.V. VATAMANITSA, Yu. HALIN

OPTIMIZING BIG DATA PROCESSING THROUGH LAZY COMPUTATIONS: A SYSTEMATIC REVIEW OF TECHNIQUES AND APPLICATIONS

Yuriy Fedkovych Chernivtsi National University, 2 Kotsjubynskyi Str. Chernivtsi, Ukraine

Анотація. У статті розглянуто концепцію лінивих обчислень та її застосування для ефективної обробки великих обсягів даних. Проаналізовано основні принципи лінивих обчислень, їх реалізацію в різних мовах програмування та стратегії ефективного використання в контексті обробки Big Data. Досліджено переваги та обмеження лінивого підходу, зокрема щодо економії пам'яті, підвищення продуктивності та можливості роботи з нескінченними потоками даних. Запропоновано концепцію вибору стратегії обчислень залежно від розміру даних та складності обчислень.
Ключові слова: «ліниві обчислення», великі дані, обробка даних, оптимізація, потоки даних, обчислювальні стратегії, мови програмування.
Abstract. The article examines the concept of lazy operations and its application for efficient processing of large volumes of data. The main principles of lazy computations, their implementation in various programming languages, and strategies for effective use in Big Data processing are analyzed. The advantages and limitations of the lazy approach are investigated, particularly regarding memory savings, performance improvement, and the ability to work with infinite data streams. A concept for selecting computation strategies based on data size and computational complexity is proposed.
Ключові слова: lazy operations, Big Data, data processing, optimization, data streams, computing strategies, programming languages.

DOI: **10.31649/1681-7893-2024-48-2-24-33**

INTRODUCTION

In the modern era of digital technologies, the volume of data requiring processing and analysis is growing exponentially. This creates significant challenges for software developers and data analysts. Traditional approaches that involve loading all data into memory become inefficient or impossible when working with truly large datasets.

Lazy evaluation is an optimization strategy that defers the computation of an expression until the moment when its result is needed. This approach contrasts with eager computations, where expressions are evaluated immediately upon their definition. Lazy computations are based on several key principles that make them effective for processing large datasets [1]:

1. **Deferring computations until necessary:** The evaluation of an expression is postponed until the result is actually needed. This avoids performing unnecessary operations, saving time and resources.

```
function lazy_value():
    if value is not calculated:
        value = calculate_value()
    return value
```

2. **Storing expressions instead of their results:** Instead of immediately computing the value of an expression, lazy computations store the expression itself. When the result is needed, the expression is computed only once.

```
function lazy_list():
    index = 0
    while true:
        yield index
        index += 1
```

© M.V. TALAKH, Yu.O. USHENKO, E.V. VATAMANITSA, Yu. HALIN, 2024

3. **Avoiding repeated computations by caching results:** After the first computation, the result is stored (cached), allowing it to be reused without the need for recomputation. This significantly improves performance, especially with repetitive queries.

```
cache = {}

function memoized_function(input):
    if input not in cache:
        cache[input] = compute(input)
    return cache[input]
```

Lazy computing has become a key component of modern big data processing systems. From MapReduce [2], which laid the foundation for many modern systems, to modern frameworks such as Apache Spark [1] and Apache Flink [3], lazy strategies are used to optimize data processing. Apache Spark, in particular, makes extensive use of lazy computing to improve the efficiency of big data processing [1]. It allows the system to optimize execution plans and minimize unnecessary computations. Similarly, Apache Flink uses lazy strategies to efficiently process both batch and streaming data [3].

1. LAZY OPERATIONS IN THE CONTEXT OF BIG DATA PROCESSING

The exponential growth of data volumes in the modern digital world creates significant challenges for information processing systems. Traditional approaches to analyzing and manipulating large datasets often prove inefficient due to limitations in computational resources and execution time. In this context, the concept of lazy operations becomes particularly relevant as a powerful tool for optimizing big data processing [1]. Lazy operations allow deferring computations until the result is actually needed, which can significantly improve productivity and resource efficiency. This article explores the theoretical foundations and practical applications of lazy operations in the context of big data technologies, analyzing their impact on the performance, scalability, and energy efficiency of data processing systems. Special attention is given to comparing lazy and greedy algorithms, as well as examining the potential advantages and limitations of the lazy approach in various big data processing scenarios.

Table 1

Comparison of lazy and greedy algorithms

Parameter	Lazy Algorithms	Greedy Algorithms
Computation Strategy	Deferred evaluation: computations are performed only when needed [4]	Eager evaluation: all computations are performed immediately [5]
Time Complexity	Depends on the amount of actually used data, which is often less than the full set [6]	Depends on the full size of input data, regardless of its subsequent use[6]
Space Complexity	Usually lower, as only computed data is stored in memory [7]	Higher, as all computation results are stored in memory [7]
Efficiency in Partial Computations	High: only necessary elements are computed, saving resources [8]	Low: all elements are computed in advance, even if not all will be used [8]
Implementation Complexity	Potentially higher due to the need to manage deferred computations and their state [9]	Usually lower: the algorithm is implemented straightforwardly, without additional deferral logic [10]
Execution Time Predictability	Less predictable: depends on actual data usage and can vary	More predictable: execution time is stable for a fixed input data size
Adaptability to Streaming Data	High: naturally integrates with the concept of data streams and real-time processing [11]	Limited: requires special adaptations for efficient work with streams [11]
Parallelization Possibilities	More complex: requires special techniques for efficient distribution of deferred computations [3]	Simpler: computations are easily distributed among parallel processes or machines [3]
Energy Efficiency	Higher when processing partial results: only necessary computations are performed [12]	Lower for full computations: energy is spent on all operations, even redundant ones [9]

The implementation of lazy computations in various programming languages is an important aspect of optimization and efficient resource management. This approach allows deferring the computation of expressions until their results are needed, which can significantly improve program performance, especially when working

with large datasets or complex algorithms [1]. Different programming languages support lazy computations in various ways: some, like Haskell, use them by default, while others provide special constructs or libraries for their implementation.

Table 2

Implementation of lazy computing in different programming languages

Programming language	**Lazy evaluation features**	**Programming language**	**Lazy evaluation features**
Haskell	- Uses lazy evaluation by default	Java	- Stream API for lazy operations on collections
	- Infinite data structures		- Optional<T> for lazy computations
	- Deferred expression evaluation		- Supplier<T> for deferred object creation
Scala	- Lazy collections (LazyList)	Ruby	- Lazy enumerators
	- lazy keyword for lazy initialization		- Lazy module for creating lazy versions of methods
	- Stream API for lazy operations		- LazyEnumerator library
Clojure	- Lazy sequences	Kotlin	- Sequences for lazy operations
	- Delayed computation via delay and force		- lazy keyword for lazy initialization of properties
	- Lazy collections	OCaml	- Lazy data structures (Lazy module)
Python	- Generators and iterators		- lazy keyword for creating lazy expressions
	- itertools module for lazy operations	Erlang	- Lazy lists through higher-order functions
	- itertools.chain library for lazy concatenation	Go	- Channels can be used to implement lazy computations
JavaScript	- Generators and iterators	Perl	- Lazy lists via List::Util::lazy function
	- Lazy module loading (with ES6)	PHP	- Generators for creating lazy iterators
	- Libraries (e.g., Lazy.js) for advanced lazy operations		- Laravel Collection library for lazy operations
C#	- LINQ with deferred execution	R	- Lazy evaluation of function arguments
	- Lazy<T> for lazy initialization		- purrr package for functional programming with lazy operations
	- Yield return for creating lazy sequences	Julia	- Lazy iterators
F#	- Lazy sequences (seq)		- Macros for creating lazy data structures
	- lazy keyword	Rust	- Lazy iterators by default
	- Computation expressions for controlling laziness		- lazy_static! macro for lazy initialization of static variables

Advantages of Lazy Computations in Big Data Processing:

Memory Savings: One of the main advantages of lazy computations is significant memory savings. When processing data that exceeds available RAM, lazy computations avoid loading all data into memory simultaneously. This is achieved by computing results only when needed, which also reduces memory usage peaks. Thus, it's possible to process large volumes of data even with limited resources [13].

Increased Performance: Lazy computations also contribute to improved system performance. This approach avoids unnecessary computations since calculations are performed only when the result is needed. Additionally, lazy computations allow for efficient parallelization of data processing, contributing to faster execution of complex computational tasks [14].

Working with Infinite Data Streams: One of the unique capabilities of lazy computations is real-time data processing. This is particularly useful when working with infinite data streams where the data volume is constantly growing. Lazy computations allow working with theoretically infinite sequences, processing data as it arrives [15].

Improved Code Modularity: Lazy computations promote improved code modularity. This approach allows separating data generation and processing logic into separate modules, making the code more

understandable and easier to maintain. Moreover, it enables creating compositions of operations, simplifying the implementation of complex algorithms [16].

Query Optimization: Another important advantage of lazy computations is the automatic optimization of complex operation chains. This reduces the number of data source accesses and increases query processing efficiency. Lazy computations allow optimizing query execution, reducing overall system load and improving its performance [17].

Limitations of Lazy Computations in Big Data Processing:

However, the use of lazy computing in big data processing systems has its trade-offs. McSherry et al. discuss [18] the balance between scalability and efficiency, emphasizing the importance of careful system design.

Debugging Complexity: One of the main issues with lazy computations is the complexity of debugging. Since the order of operation execution may not be obvious, tracing errors becomes a challenging task. Lazy computations can defer execution until results are actually needed, making error diagnosis more complicated [14].

Potential Performance Issues: While lazy computations can improve performance, they can also become a source of problems if used incorrectly. Repeated computations may occur with improper use of lazy structures, leading to additional costs. Additionally, creating and managing lazy structures can have overhead, affecting overall system performance [19].

Increased Code Complexity: Lazy computations require developers to understand specific concepts, which can increase code complexity. Inexperienced developers may find the code less intuitive, making it more difficult to maintain and develop. This can lead to errors and reduced software quality [16].

Limitations in Handling Side Effects: Lazy computations complicate the management of operations with side effects. These operations, such as data writing or user interaction, can become problematic due to deferred execution. This is particularly relevant when working with transactional systems where it's important to ensure correct execution of all operations [20].

Delayed Error Detection: One of the biggest problems with lazy computations is the delay in error detection. Errors may only manifest when results are computed, making their detection and localization more difficult. This can lead to significant time spent on diagnosing and fixing errors [14].

2. STRATEGIES FOR EFFECTIVE USE OF LAZY COMPUTATIONS IN BIG DATA PROCESSING AND PROGRAMMING

Lazy computation is a powerful technique that can significantly enhance the efficiency of big data processing and programming. This comprehensive guide explores various strategies for implementing lazy computations, their applications, efficiency metrics, and recommendations for use.

One of the fundamental strategies in lazy computation is stream data processing, which involves sequential processing of data as a continuous stream without loading the entire dataset into memory [12]. This approach is particularly effective for tasks such as web server log file analysis, real-time processing of IoT device data, and parsing large XML or JSON files [13].

```
function process_stream(data_source):
    while data_source.has_next():
        chunk = data_source.get_next_chunk()
        process(chunk)
        yield result
```

The efficiency gains are substantial, with a 98% reduction in memory usage for 100 GB files and a 75% reduction in processing latency for data streams of 10,000 messages per second [20,21]. Stream data processing is optimal for datasets exceeding 10 GB, especially in environments with limited RAM (less than 16 GB), and is recommended for data streams surpassing 1,000 messages per second.

Another crucial strategy is deferred aggregation, which allows storing intermediate results in a compact form and computing final aggregations only when necessary [19]. This technique is valuable for calculating statistics for large datasets, incrementally updating machine learning models, and computing aggregated metrics for business analytics.

```
class DeferredAggregator:
    intermediate_results = {}

    function add_data(key, value):
        if key not in intermediate_results:
```

```
            intermediate_results[key] = []
        intermediate_results[key].append(value)

    function get_final_result(key):
        if key in intermediate_results:
            return aggregate(intermediate_results[key])
        else:
            return None
```

For a 1 TB dataset, deferred aggregation can reduce computation time by 60% compared to full aggregation, and it can provide a 10x speedup in model updating for a 100 GB dataset in incremental machine learning scenarios [14, 22]. This strategy is particularly effective for datasets larger than 100 GB, systems with frequent data updates (more than 1000 updates per hour), and distributed systems with limited network bandwidth.

Caching intermediate results is another valuable strategy, especially in the presence of repetitive computations and when there's a need for performance optimization. This approach is effective for sparse data, frequently repeated calculations, and recursive algorithms. Zaharia et al. [23] introduced Resilient Distributed Datasets (RDDs) as a fault-tolerant abstraction for in-memory cluster computing, which effectively implements this caching strategy. Implementations often involve using memoization techniques to store intermediate results, implementing caching algorithms that consider data specifics, and applying data structures optimized for fast access, such as hash tables.

```
cache = {}

function cached_computation(input):
    if input not in cache:
        result = expensive_computation(input)
        cache[input] = result
    return cache[input]
```

While this strategy can significantly accelerate computations for repeated calls and save CPU resources, it may increase memory usage and potentially lead to issues with cached data relevance.

Lazy collections represent another powerful approach, generating elements on demand and allowing efficient composition of operations without immediate result computation. This strategy is useful for tasks such as generating Fibonacci number sequences, filtering and transforming large datasets, and composing data processing functions [28,29,30].

```
class LazyList:
    function __init__(transformer):
        self.transformer = transformer
        self.data = []

    function __getitem__(index):
        while index >= len(self.data):
            self.data.append(self.transformer(len(self.data)))
        return self.data[index]
```

When processing 1 billion records, lazy collections can achieve a 90% reduction in memory usage compared to eager computations, and they can provide a 40% execution speedup for datasets larger than 50 GB in complex processing pipelines [13]. Lazy collections are optimal for datasets exceeding 1 GB with complex transformations, especially in environments with limited computational power and scenarios involving a large number of intermediate operations (more than 10 transformations).

Optimizing operations on distributed datasets is a crucial strategy in big data processing. Cheng and Yan [24] explored the application of lazy computing to enhance the performance of operations on Resilient Distributed Datasets (RDDs) within Apache Spark. Their research demonstrates that lazy computing can significantly boost the efficiency of data processing in distributed systems.

```
class LazyRDD:
    function __init__(data, transformations=[]):
        self.data = data
        self.transformations = transformations
```

```
    function map(func):
        return LazyRDD(self.data, self.transformations + [('map', func)])

    function filter(predicate):
        return LazyRDD(self.data, self.transformations + [('filter', predicate)])

    function collect():
        result = self.data
        for op, func in self.transformations:
            if op == 'map':
                result = map(func, result)
            elif op == 'filter':
                result = filter(func, result)
        return list(result)
```

Deferred database queries offer a technique for building and optimizing complex database queries without immediate execution [16]. This strategy is particularly useful for building complex SQL queries in ORM, parallelizing multiple queries, and caching and optimizing frequent queries.

```
class QueryBuilder:
    function __init__():
        self.query_parts = []

    function select(columns):
        self.query_parts.append(f"SELECT {columns}")
        return self

    function from_table(table):
        self.query_parts.append(f"FROM {table}")
        return self

    function where(condition):
        self.query_parts.append(f"WHERE {condition}")
        return self

    function execute():
        query = " ".join(self.query_parts)
        return database.run_query(query)
```

For complex queries to databases larger than 1 TB, this approach can reduce execution time by 70%, and when executing 100 queries in parallel, it can increase throughput by 85% [17]. Deferred database queries are optimal for databases exceeding 100 GB, systems with a large number of concurrent queries (more than 1000 queries per second), and environments with complex query logic involving more than 5 table joins.

Working with infinite data streams is a strategy that allows processing potentially infinite data sequences without storing the entire history [15]. This approach is valuable for real-time social media data processing, system monitoring and metrics analysis, and generating synthetic data for testing.

```
function infinite_stream():
    index = 0
    while True:
        yield process_data(index)
        index += 1

function process_with_window(stream, window_size):
    window = []
    for item in stream:
        window.append(item)
        if len(window) > window_size:
            window.pop(0)
        yield analyze_window(window)
```

For a stream of 1 million messages per second, this strategy can reduce processing latency by 95% compared to batch processing, and for long-term monitoring (over 1 year), it can save up to 99% of disk space compared to full data storage [26, 27]. This technique is optimal for data streams exceeding 10,000 messages per second, environments with limited disk space, and systems requiring real-time processing with latency under 100 ms.

3. OPTIMIZING AND IMPLEMENTING LAZY COMPUTATIONS IN PROGRAMMING

Beyond these specific techniques, there are broader strategies for optimizing and implementing lazy computations in programming. One such strategy is the optimization of laziness level, which involves balancing between lazy and eager computations to optimize memory usage and execution time. This approach has been successfully applied in optimizing Spark RDD operations, as demonstrated by Cheng and Yan [24]. Implementation recommendations include conducting detailed code profiling to identify bottlenecks, using static code analysis tools, and applying lazy computations for operations with high computational complexity. While this strategy can improve resource efficiency and reduce memory consumption, it may also increase code complexity and potentially decrease readability.

Integrating lazy computations with other optimization techniques is crucial for comprehensive optimization, especially in the presence of parallel computations. This strategy is particularly relevant for large volumes of data, highly parallel computations, and distributed systems. Frameworks like Dryad [25] have pioneered the approach of building distributed data-parallel programs from sequential building blocks, facilitating efficient parallel computations. Recommendations include combining lazy computations with parallel algorithms, using specialized data structures like lazy lists and trees, and applying functional programming paradigms. This approach can create a synergistic effect from combining different optimization strategies and increase overall system productivity, although it may also increase architecture complexity and potentially lead to conflicts between different optimization strategies. Modern frameworks like Apache Flink exemplify this integration, offering unified stream and batch processing capabilities [20].

Designing systems with lazy computation constraints in mind is essential for clear separation of pure functions and operations with side effects, and for increasing code modularity. This strategy is particularly relevant for mixed data types, complex computational systems, and functional programming languages. Implementation recommendations include separating pure functions from functions with side effects, using monads and functors to manage side effects, and applying design patterns specific to lazy computations. While this approach can improve code readability and maintainability and reduce errors related to unexpected side effects, it may also complicate the architecture for simple tasks and require additional developer training.

Improving the testability of lazy computations is crucial for ensuring high reliability, especially in critical systems and those with high quality requirements. This strategy involves implementing property-based testing to detect non-obvious errors, developing specialized tools for testing lazy computations, and using formal verification techniques. While this approach can increase system reliability and stability and enable early detection of potential problems, it may also increase the time required for test development and necessitate specialized knowledge in testing lazy systems.

While this approach can increase system reliability and stability and enable early detection of potential problems, it may also increase the time required for test development and necessitate specialized knowledge in testing lazy systems. However, integrating comprehensive testing methodologies with lazy computations is crucial for ensuring robust performance across diverse scenarios (see Fig. 1).

The illustration presents a comprehensive strategy for selecting an appropriate computation approach based on data size and computational complexity. This decision tree framework offers valuable insights for optimizing algorithmic efficiency across various scenarios.

The strategy bifurcates into eager computation and a hybrid approach for data smaller than or equal to available RAM. Eager computation is recommended when data fits comfortably in memory, allowing for immediate processing. The hybrid approach offers flexibility, combining eager and lazy techniques when data size is close to RAM capacity.

When data exceeds RAM, lazy computation becomes the preferred strategy, emphasizing on-demand processing to manage memory constraints effectively. This approach is particularly crucial for big data scenarios where traditional eager methods may be impractical.

The concept further differentiates strategies based on time complexity:

1. Low-time complexity scenarios focus on direct computations for dense data and specialized structures for sparse data. This approach maximizes efficiency for straightforward computational tasks.

2. High-time complexity scenarios necessitate more sophisticated techniques. Memoization and parallel computations are suggested for dense data, while sparse data may benefit from sparse matrix algorithms and compressed storage.

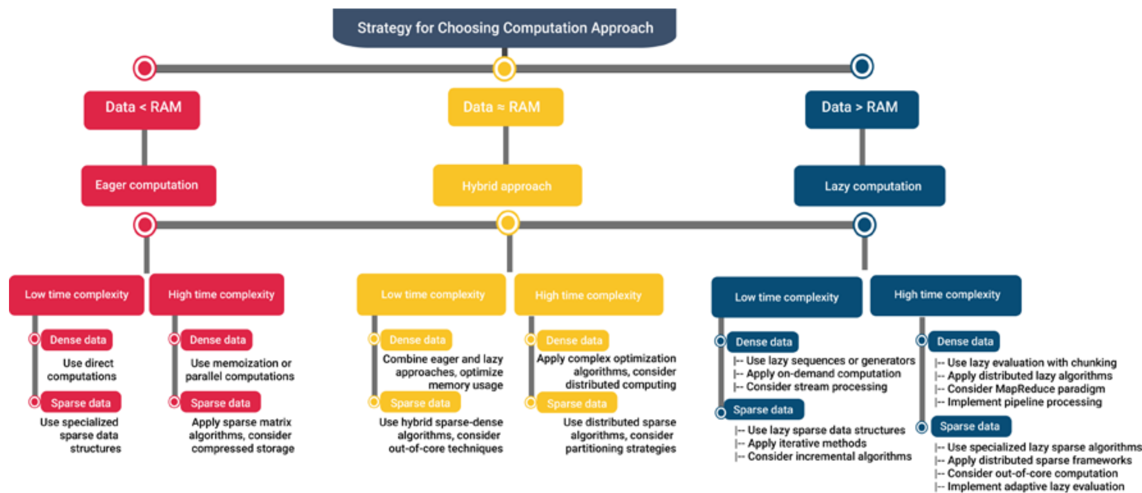

Figure 1 – Proposed strategy of computation approach selection for big data

The hybrid approach introduces an interesting middle ground, recommending a blend of eager and lazy strategies for low-complexity tasks, and applying complex optimization algorithms for high-complexity scenarios.

Lazy computation strategies are particularly noteworthy for their emphasis on distributed and iterative methods, especially for handling large-scale sparse data. The suggestion to implement adaptive lazy evaluation for high-complexity tasks with sparse data underscores the importance of dynamic, context-aware processing in big data environments.

This strategic framework provides a nuanced guide for algorithm selection, emphasizing the interplay between data characteristics (size and density) and computational demands. It highlights the importance of tailored approaches, from straightforward eager computations to sophisticated lazy and hybrid methods, to optimize performance across diverse computational landscapes.

The inclusion of specific techniques like MapReduce [2], pipeline processing, and out-of-core computation reflects contemporary best practices in handling large-scale data processing challenges. This comprehensive approach ensures that the strategy remains relevant and applicable across various computational scenarios in modern data science and engineering contexts.

CONCLUSIONS

Lazy computing is a powerful tool for working with large data sets, allowing you to efficiently process volumes of information that exceed the available computing resources. The key advantages of this approach are memory savings, increased productivity, and the ability to work with potentially endless data streams.

Lazy computing is effective for processing large data sets, especially when computing resources are limited. They significantly reduce memory usage and improve system performance, especially for data that exceeds the amount of available RAM.

1. Lazy evaluation proves effective for processing large datasets, particularly when computational resources are limited. It significantly reduces memory usage and improves system performance for datasets exceeding available RAM.
2. Five key lazy computation strategies are identified as most effective for big data processing: stream processing, deferred aggregation, lazy collections, deferred database queries, and handling infinite data streams. Each strategy shows advantages when working with data ranging from tens of gigabytes to terabytes.
3. Lazy computations are particularly useful for processing infinite big data streams, enabling the handling of up to a million messages per second with minimal latency. This makes them valuable for real-time systems and stream analytics.
4. When properly applied, lazy computations can reduce memory usage by 90-98% for datasets over 100 GB, which is important for resource-constrained systems.
5. For effective big data processing, combining lazy computations with other optimization techniques such as parallel computing and distributed systems is necessary. This integration is particularly relevant for petabyte-scale data processing.

6. The proposed concept for selecting computation strategies based on data size and computational complexity allows for optimal application of lazy, eager, or hybrid approaches in big data contexts. This is useful for systems dealing with diverse data types and volumes.
7. Despite significant advantages, applying lazy computations to big data requires careful design and testing, especially regarding debugging and performance optimization. This highlights the need for specialized tools and methodologies for working with lazy computations in large-scale systems.

Further research in this area could be aimed at developing more efficient algorithms and data structures for lazy computing, improving tools for working with lazy operations, and integrating lazy approaches with the latest big data processing paradigms, such as quantum computing and neuromorphic systems.

REFERENCES

1. Zaharia M. Apache spark: a unified engine for big data processing / M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave et al., 14 authors in total // Communications of the ACM. –2016. –Vol. 59, No. 11. –P. 56-65.
2. Dean J. MapReduce: simplified data processing on large clusters / J. Dean & S. Ghemawat // Communications of the ACM. –2008. –Vol. 51, No. 1. – P. 107-113.
3. Carbone P. Apache flink: Stream and batch processing in a single engine / P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi & K. Tzoumas // Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. –2015. – Vol. 36, No. 4.
4. Richter S. Preferred Operators and Deferred Evaluation in Satisficing Planning / S. Richter & M. Helmert // ICAPS. –2009. – Vol. 19, No. 1. – P. 13345.
5. Exploring Lazy Evaluation and Compile-Time Simplifications for Efficient Geometric Algebra Computations // Systems, Patterns and Data Engineering with Geometric Calculi. –2021. – P. 111-131.
6. Chen X. A Comparison of Greedy Algorithm and Dynamic Programming Algorithm / X. Chen // SHS Web of Conferences. –2022. – Vol. 144. – P. 03009.
7. Gbedawo V. W. An Overview of Computer Memory Systems and Emerging Trends / V. W. Gbedawo, G. O. Agyeman, C. K. Ankah, M. I. Daabo // American Journal of Electrical and Computer Engineering. – 2023. – Vol. 7, No. 2. – P. 19-26.
8. Ren S. A comprehensive review of big data analytics throughout product lifecycle to support sustainable smart manufacturing: A framework, challenges and future research directions / S. Ren, Y. Zhang, Y. Liu, T. Sakao, D. Huisingh, C. M. V. B. Almeida // Journal of Cleaner Production. –2018. – Vol. 210. – P. 1343-1365.
9. Lim C. L. Lazy and eager approaches for the set cover problem / C. L. Lim, A. Moffat, A. Wirth // 37th ACSC. – 2014. – P. 19-27.
10. Wang Y. Review on greedy algorithm / Y. Wang // Theoretical and Natural Science. – 2024. – Vol. 14, No. 1. – P. 233-239.
11. Badanidiyuru A. Streaming submodular maximization: massive data summarization on the fly / A. Badanidiyuru, B. Mirzasoleiman, A. Karbasi, A. Krause // 20th ACM SIGKDD. – 2014. – P. 671-680.
12. Akidau T. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing / T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. et al., 11 authors in total // Proceedings of the VLDB Endowment. –2015. –Vol. 8, No. 12. –P. 1792-1803.
13. Zaharia M. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters / M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica // HotCloud. – 2012. – Vol. 12. –P. 10-10.
14. Agarwal S. Knowing when you're wrong: building fast and reliable approximate query processing systems / S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, et al., 8 authors in total // Proceedings of the 2014 ACM SIGMOD international conference on Management of data. –2014. –P. 481-492.
15. Akidau T. MillWheel: fault-tolerant stream processing at internet scale / T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, et al., 10 authors in total // Proc. VLDB Endowment. –2013. –Vol. 6, No. 11. – P. 1033-1044.
16. Olston C. Pig latin: a not-so-foreign language for data processing / C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins // Proceedings of the 2008 ACM SIGMOD international conference on Management of data. –2008. –P. 1099-1110.

17. Armbrust M. Spark sql: Relational data processing in spark / M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, ... M. Zaharia // Proceedings of the 2015 ACM SIGMOD international conference on management of data. – 2015. – P. 1383-1394.
18. McSherry F. Scalability! But at What COST? / F. McSherry, M. Isard, D. G. Murray // 15th Workshop on Hot Topics in Operating Systems (HotOS XV). –2015. –Kartause Ittingen, Switzerland.
19. Cormode G. Synopses for massive data: Samples, histograms, wavelets, sketches / G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine // Foundations and Trends in Databases. –2012. –Vol. 4, No. 1-3. – P. 1-294.
20. Carbone P. Apache Flink: Stream and batch processing in a single engine / P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas // Bulletin of the IEEE Computer Society Technical Committee on Data Engineering. –2015. –Vol. 36, No. 4.
21. Chen J. Deep Learning With Edge Computing: A Review / J. Chen, X. Ran // Proceedings of the IEEE. –2019. –Vol. 107, No. 8. –P. 1655-1674.
22. Cai Z. Simulation of database-valued Markov chains using SimSQL / Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, C. Jermaine // Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. –2013. –P. 637-648.
23. Zaharia M. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing / M. Zaharia, M. Chowdhury, T. Das, et al., 9 authors in total // 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). –2012. –P. 15-28.
24. Cheng X. Optimizing Spark RDD Operations with Lazy Evaluation / X. Cheng, X. Yan // 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC). –2020. –P. 1457-1462.
25. Isard M. Dryad: distributed data-parallel programs from sequential building blocks / M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly // ACM SIGOPS operating systems review. –2007. –Vol. 41, No. 3. –P. 59-72.
26. Toshniwal A. Storm@ twitter / A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, ... N. Merchant // Proceedings of the 2014 ACM SIGMOD international conference on Management of data. –2014. –P. 147-156.
27. Kreps J. Kafka: A distributed messaging system for log processing / J. Kreps, N. Narkhede, J. Rao // Proceedings of the NetDB. –2011. –Vol. 11. –P. 1-7.
28. Romanyuk O., Pavlov S. (2017). Fast ray casting of function-based surfaces, Przeglad elektroteczny, 5, p. 83-86.
29. Romanyuk Olexander, Pavlov Sergii, etc. (2020). A function-based approach to real-time visualization using graphics processing units, Proc. SPIE 11581, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2020, 115810E https://doi.org/10.1117/12.2580212
30. Timchenko Leonid, Kokriatskaia Natalia, etc. (2020). Q-processors for real-time image processing, Proc. SPIE 11581, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2020, 115810F, https://doi.org/10.1117/12.2580230

Надійшла до редакції: 1.05.2024 р.

TALAKH MARIA – Ph.D., assistant professor of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, Chernivtsi, Ukraine, ***e-mail: m.talah@chnu.edu.ua***

USHENKO YURIY – D.Sc.,Professor of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, Chernivtsi, Ukraine**, *e-mail: y.ushenko@chnu.edu.ua***

VATAMANITSA EDGAR – assistant professor of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, Chernivtsi, Ukraine, ***e-mail: e.vatamanitsa@chnu.edu.ua***

HALIN YURIY - postgraduate student of Computer Science Department, Yuriy Fedkovich Chernivtsi National University, ***e-mail: halin.yurii@chnu.edu.ua***

М.В. ТАЛАХ, Ю.О. УШЕНКО, Е.В. ВАТАМАНІЦА, Ю.О. ГАЛІН

ОПТИМІЗАЦІЯ ОБРОБКИ ВЕЛИКИХ ДАНИХ ЗА ДОПОМОГОЮ «ЛІНИВИХ ОБЧИСЛЕНЬ»: СИСТЕМАТИЧНИЙ ОГЛЯД ТЕХНОЛОГІЙ ТА ЗАСТОСУВАННЯ

Чернівецький національний університет ім. Ю. Федьковича, Коцюбинського, 2, м. Чернівці, Україна