УДК 004.421:004.738.5

Изрипов Юсуп Заирбекович

Ведущий разработчик

Т-банк, г. Грозный, Чеченская Республика, Российская Федерация

ОПТИМИЗАЦИЯ ВЫЧИСЛИТЕЛЬНОЙ СЛОЖНОСТИ ОПЕРАЦИЙ ВО ФРОНТЕНДЕ: ЗАМЕНА ЛИНЕЙНЫХ СТРУКТУР ДАННЫХ ХЕШ-ТАБЛИЦАМИ ДЛЯ ДОСТИЖЕНИЯ ОЖИДАЕМОГО КОНСТАНТНОГО ВРЕМЕНИ ДОСТУПА ПО КЛЮЧУ

Аннотация. Современные фронтенд-приложения обрабатывают крупные коллекции сущностей и многократно выполняют операции поиска и обновления данных по идентификатору. Последовательный просмотр массива делает стоимость поиска зависимой от размера коллекции, тогда как хеш-индексация позволяет снизить ожидаемую стоимость повторного доступа по ключу. Цель исследования состоит в разработке критерия выбора между линейным поиском и хеш-структурой с учетом стоимости построения индекса и числа повторных обращений. Микробенчмарк Array.find() и Map.get() для коллекций от 1000 до 100000 объектов показал, что для наборов от 10000 до 100000 объектов порог окупаемости составил около 230–256 обращений, а при 1000 поисков в коллекции из 100000 объектов измеренное время хеш-сценария оказалось приблизительно на 74,9% ниже. Полученные результаты показывают, что эффективность замены структуры определяется размером коллекции, частотой доступа, временем существования индекса и характером обновления данных.

Ключевые слова: производительность фронтенда; вычислительная сложность; структуры данных; хеш-таблица; JavaScript; Map; Array; оптимизация поиска.

OPTIMIZATION OF THE COMPUTATIONAL COMPLEXITY OF FRONTEND OPERATIONS: REPLACING LINEAR DATA STRUCTURES WITH HASH TABLES FOR EXPECTED CONSTANT-TIME KEY ACCESS

Yusup Izripov

Lead Developer
T-Bank, Grozny, Chechen Republic, Russian Federation
ORCID: …

Abstract. Modern frontend applications process large collections of entities and repeatedly perform search and update operations by identifier. Sequential array scanning makes lookup cost dependent on collection size, whereas hash indexing can reduce the expected cost of repeated key-based access. The aim of this study is to develop a criterion for choosing between linear search and a hash-based structure while accounting for index construction cost and the number of repeated accesses. A microbenchmark of Array.find() and Map.get() for collections ranging from 1,000 to 100,000 objects showed that, for datasets of 10,000 to 100,000 objects, the break-even threshold was approximately 230–256 accesses, while for 1,000 lookups in a collection of 100,000 objects, the measured execution time of the hash-based scenario was approximately 74.9% lower. The results indicate that the effectiveness of replacing the data structure depends on collection size, access frequency, index lifetime, and the data update pattern.

Keywords: frontend performance; computational complexity; data structures; hash table; JavaScript; Map; Array; lookup optimization.

Introduction

Modern frontend applications process catalogs, user lists, messages, access permissions, cached responses, and other collections of entities. In many scenarios, a particular object must be retrieved by its identifier multiple times within a single user action or state update cycle. When a collection is represented solely as an array, calls to Array.find(), Array.findIndex(), and similar operations require a sequential scan of the elements each time. Studies of the dynamic behavior of JavaScript programs indicate that real-world applications make extensive use of objects and arrays, and that data access patterns have a substantial impact on actual program execution [1]. This issue becomes increasingly significant as collection size and the frequency of repeated accesses grow. JavaScript performance measurements confirm that localized implementation changes can yield substantial performance improvements, although

the magnitude of the gain depends on the specific workload and execution environment [2]. Therefore, the choice of data structure should be guided by the measured operation profile rather than by the theoretical advantage of one structure over another alone.

Objectives and research tasks

The aim of this study is to develop and experimentally validate a methodological approach for determining when repeated linear search should be replaced with access through a hash-based structure.

The research objective is to apply a measurable criterion for selecting a data structure. The proposed criterion is the threshold number of accesses q*, which accounts for the cost of index construction and the actual costs associated with the two access methods. The research question is formulated as follows: at what number of repeated accesses does the reduction in subsequent lookup time offset the cost of constructing the index? To answer this question, we use asymptotic analysis, a break-even model of total costs, and a microbenchmark in a JavaScript environment.

Methods

Arrays retain their advantages for sequential iteration, sorting, filtering, and preserving display order. The situation differs for key-based lookup: for a collection of n elements, a sequential scan has a complexity of O(n), while q independent lookups result in an overall complexity of O(qn). In studies of JavaScript data structures, hash tables are regarded as a specialized mechanism for providing fast key-based access [3]. A hash-based structure maps a key to a storage location using a hash function. When properly implemented, the expected lookup cost approaches O(1). The dynamic perfect hashing model demonstrates the possibility of constant-time lookup with linear space requirements and expected amortized constant-time updates under specified assumptions [4]. Cuckoo hashing provides an alternative key-placement strategy and likewise enables constant-time lookup under certain conditions [5]. In JavaScript, Map provides a practical mechanism for indexed access. Initially constructing the structure from n elements requires O(n) time, after which repeated key-based access can have an expected constant-time cost. In this

context, O(1) describes the expected performance characteristic of hash-based access. However, asymptotic complexity alone does not determine the actual efficiency of switching to Map. The ECMAScript standard does not require Map access operations to have constant-time complexity; rather, implementations must, on average, provide access times that are sublinear with respect to the number of elements in the collection [6]. Thus, the practical assumption of expected O(1) performance depends on the specific engine implementation and the indexing mechanism employed. This distinction is essential for practical optimization because the theoretical advantage of a data structure must be distinguished from the measurable performance gain in a given execution environment. In V8, Map and Set are implemented using hash-table mechanisms; however, the internal representation of keys, storage of hash codes, and related optimizations have changed across engine versions [7]. Consequently, the same JavaScript construct may exhibit different constant factors while retaining the same asymptotic complexity class.

The optimization approach proposed in this study consists of five stages (Fig. 1):

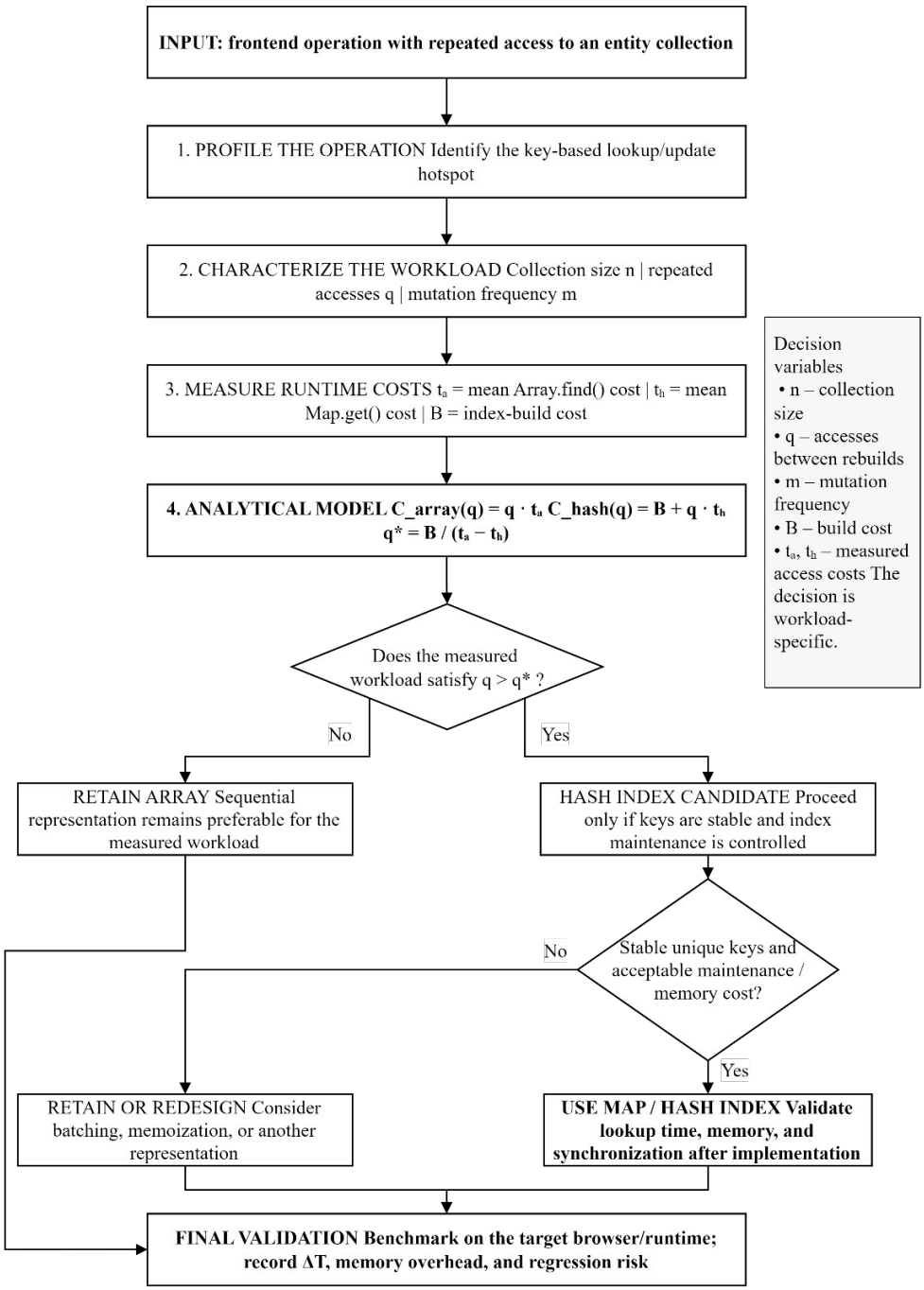

Figure 1 – Method for selecting a data structure for repeated key-based access

Source: compiled by the author.

The experiment was conducted in Node.js 22.16.0 using the V8 engine on an Intel Xeon Platinum 8272CL 2.60 GHz processor. The collections consisted of objects with a numeric id field; for each collection size, 1,000 deterministic accesses were performed using uniformly distributed keys. The linear-search scenario used Array.find(), whereas in the hash-based scenario, a Map was constructed through

successive calls to set(), followed by Map.get() operations. Ten warm-up runs were performed before measurement, and the final result was calculated as the median of 31 measurements. Rendering, DOM operations, and network operations were excluded from the test. The reproducibility of the microbenchmark warrants particular attention. In managed runtime environments, the results of short benchmarks are affected by virtual machine warm-up, JIT compilation, garbage collection, and background system activity. Methodological studies of performance evaluation in managed runtime environments demonstrate the need for repeated runs and statistical aggregation, as a single measurement may reflect a transient state of the environment rather than a stable property of the implementation under study [8, 9]. Research on virtual machine warm-up dynamics further indicates that reaching a steady state does not necessarily occur consistently even for small deterministic programs [10]. In the present experiment, this limitation was partially addressed by performing ten warm-up runs and using the median of 31 measurements. The median reduces the influence of individual outliers but does not eliminate the dependence of the results on the V8 version, hardware platform, and JIT compiler behavior. Therefore, the reported values should be interpreted as estimates of the threshold for the specified configuration.

The following expressions are used for the analytical comparison of total costs:

$$C_{array}(q) = q \cdot t_a \quad (1)$$

$$C_{hash}(q) = B + q \cdot t_h \quad (2)$$

$$q^{¿} = \frac{B}{t_a - t_h}, t_a > t_h \quad (3)$$

The threshold q* is determined by the condition $C_{array(q)} = C_{hash(q)}$ and represents the minimum number of repeated accesses after which the reduction in lookup costs offsets the cost of constructing the index. The value of q refers to the lifetime of a single index; that is, the more accesses are performed without completely rebuilding the index, the more effectively the cost B is amortized.

Results

The total costs of the linear and hash-based approaches follow different patterns. For an array, the cost increases in proportion to the number of repeated accesses, whereas the hash-based approach involves a one-time cost for index construction and a comparatively low cost for subsequent accesses. The threshold q* corresponds to the point at which the costs of the two approaches are equal and provides a link between asymptotic analysis and empirical measurements (Table 1).

Table 1 – Microbenchmark results and break-even threshold calculation

n	**Array.find(), 1000 accesses, ms**	**Map build, ms**	**Map.get(), 1000 accesses, ms**	**q***
1 000	1,54	0,06	0,03	≈ 40
10 000	14,19	3,62	0,05	≈ 256
50 000	68,40	15,73	0,06	≈ 230
100 000	140,61	35,14	0,13	≈ 250

The microbenchmark showed a consistent increase in Array.find() execution time as the collection size grew. For n = 1,000, the median time for 1,000 lookups was 1.54 ms; for 10,000 objects, 14.19 ms; for 50,000 objects, 68.40 ms; and for 100,000 objects, 140.61 ms. The time required for 1,000 Map.get() accesses ranged from 0.03 to 0.13 ms. Building the Map required 0.06, 3.62, 15.73, and 35.14 ms, respectively. The calculated q* values were approximately 40 accesses for n = 1,000, 256 for 10,000, 230 for 50,000, and 250 for 100,000 objects. Thus, for large collections, several hundred repeated accesses were sufficient to offset the initial cost of index construction in the experimental environment. For n = 100,000, the complete hash-based scenario with 1,000 accesses required approximately 35.27 ms, compared with 140.61 ms for linear search. The measured execution time was reduced by approximately 74.9%, with the linear-search scenario taking approximately 3.99 times as long as the hash-based scenario (see Fig. 2).

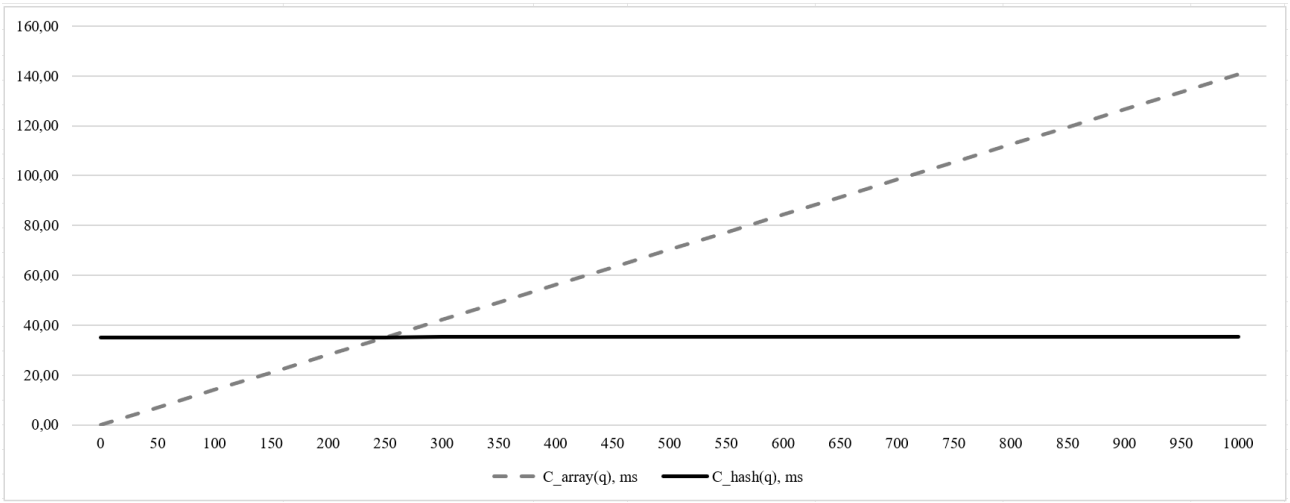

Figure 2 – Break-even model for linear search and hash-based access

Source: compiled by the author.

The value of q* did not increase in proportion to the collection size: for n = 10,000, 50,000, and 100,000, it was approximately 256, 230, and 250 accesses, respectively. Thus, collection size alone does not determine when a transition to hash indexing becomes appropriate. In particular, the threshold depends on the relationship between the cost of index construction and the difference in per-access costs between the two lookup methods. These values also allow q* to be interpreted as an indicator of how sensitive the decision is to the cost of index construction. The greater the difference between t_a and t_h, the faster the initial indexing cost is recovered; as this difference decreases, the threshold increases. Consequently, an increase in n affects the decision indirectly, through changes in the actual cost of linear search, Map construction, and subsequent access. This explains the absence of a monotonic increase in q* in Table 1. For the three largest datasets (10,000, 50,000, and 100,000 objects), the threshold falls within a relatively narrow range of 230–256 accesses, although the absolute time required for 1,000 linear searches increases from 14.19 to 140.61 ms. This result indicates that the practical criterion for switching data structures should account not for collection size in isolation, but for the relationship between one-time and recurring costs. In an applied system, this yields a testable rule: hash indexing can be considered a candidate for optimization when $q > q^*$, after which the decision should be validated through repeated measurements under the

target workload. The results confirm that replacing an array with a hash-based structure should be based on the operation profile. An array remains a rational representation when the application primarily iterates over the entire collection, sorts it, or preserves the order of visual output. Map becomes preferable when individual entities are repeatedly accessed by a stable key and a sufficient number of accesses occur between index rebuilds.

The actual performance of hashing is determined by dataset size, load factor, key distribution, the ratio of read to write operations, and the proportion of successful and unsuccessful lookups [11]. Therefore, the O(1) estimate reflects an expected property of the access method. Performance is also affected by the collision resolution mechanism and the density of element placement; specialized hashing schemes demonstrate that these parameters can substantially alter the practical cost of operations [12]. Another important factor is the lifetime of the constructed index. If a Map is created once and used for a large number of accesses, the cost B is amortized across all subsequent lookups. Conversely, frequent complete rebuilding of the structure increases the threshold q* and may eliminate the expected performance gain. Therefore, the decision to use hash indexing should take state updates into account.

In frontend code, an ordered array of identifiers can be maintained alongside a Map containing the corresponding objects. This approach is appropriate when order must be preserved for display purposes while repeated access by id is also required. A necessary condition is that the two data representations be updated consistently. These q* values should not be regarded as universal. The microbenchmark isolates data structure operations and does not account for the DOM, rendering, or overall user interface latency. The study is also limited to a single runtime environment and hardware configuration. Therefore, the optimized implementation should be profiled again in the target browser; evaluating the additional memory overhead remains a separate task.

The practical applicability of the results depends on how accurately the microbenchmark reproduces the actual workload profile of a frontend application.

Studies of JavaScript workloads show that synthetic benchmarks can differ substantially from real-world web applications in terms of execution patterns, event-driven behavior, and memory access characteristics [13]. A similar approach is used in V8 engineering practice, where the results of isolated benchmarks are compared with measurements from real websites and application scenarios [14]. Therefore, once a favorable q* has been identified at the level of data structure operations, a second level of validation is required: measuring the complete user scenario, including state updates, framework operations, and rendering. If lookup accounts for only a small fraction of overall latency, a substantial relative improvement in Map.get() performance may have only a limited effect on the user interface. Memory consumption represents an additional constraint. A hash index requires auxiliary data, and when an array and a Map are used simultaneously, consistency between the two representations must be maintained. At high update frequencies, synchronization costs and the additional memory management overhead may partially offset the performance gains from faster lookup. Recent research on hash-based structures also emphasizes that performance depends on the sequence of operations, key distribution, and patterns of state changes [15]. Therefore, further validation of the proposed approach should include memory measurements, string and object keys, successful and unsuccessful lookups, multiple browser engines, and scenarios with varying mutation rates.

Conclusion

Thus, the choice of data structure in frontend applications should be based on the actual patterns of access to the collection. Sequential search incurs O(n) cost per access and O(qn) for q repeated lookups. Hash indexing requires O(n) time to construct the structure, after which the expected cost of key-based access approaches O(1). The proposed threshold $q^* = B/(t_a - t_h)$ makes it possible to determine the number of accesses after which the index offsets its own construction cost. In the experiment, the threshold was approximately 230–256 accesses for collections containing 10,000 to 100,000 objects. For 1,000 lookups in a collection of 100,000 objects, the use of Map reduced the measured execution time by approximately

74.9%, or nearly fourfold. The value of q* should be regarded as a workload-specific parameter, and the proposed approach as a procedure for determining it experimentally. The practical significance of the study lies in the ability to justify changes in data structures on the basis of measurable performance gains and to reassess the decision after implementation.

References

1. Richards G., Lebresne S., Burg B., Vitek J. An Analysis of the Dynamic Behavior of JavaScript Programs // ACM SIGPLAN Notices. 2010. Vol. 45. No. 6. P. 1–12. DOI: 10.1145/1806596.1806598.
2. Selakovic M., Pradel M. Performance Issues and Optimizations in JavaScript: An Empirical Study // Proceedings of the 38th International Conference on Software Engineering. 2016. P. 61–72. DOI: 10.1145/2884781.2884829.
3. Bae S. Hash Tables // JavaScript Data Structures and Algorithms. Apress, 2019. P. 151–161. DOI: 10.1007/978-1-4842-3988-9_11.
4. Dietzfelbinger M., Karlin A., Mehlhorn K., Meyer auf der Heide F., Rohnert H., Tarjan R. E. Dynamic Perfect Hashing: Upper and Lower Bounds // SIAM Journal on Computing. 1994. Vol. 23. No. 4. P. 738–761. DOI: 10.1137/S0097539791194094.
5. Pagh R., Rodler F. F. Cuckoo Hashing // Journal of Algorithms. 2004. Vol. 51. No. 2. P. 122–144. DOI: 10.1016/j.jalgor.2003.12.002.
6. ECMA International. ECMAScript® 2026 Language Specification. ECMA-262. 17th ed. Geneva: Ecma International, 2026. Section 24.1 “Map Objects”. URL: https://ecma-international.org/wp-content/uploads/ECMA-262_17th_edition_june_2026.pdf
7. Gunasekaran S. Optimizing Hash Tables: Hiding the Hash Code // V8 Blog. 2018. January 29. URL: https://v8.dev/blog/hash-code
8. Georges A., Buytaert D., Eeckhout L. Statistically Rigorous Java Performance Evaluation // ACM SIGPLAN Notices. 2007. Vol. 42. No. 10. P. 57–76. DOI: 10.1145/1297027.1297033.

9. Kalibera T., Jones R. E. Rigorous Benchmarking in Reasonable Time // Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM ’13). 2013. P. 63–74. DOI: 10.1145/2464157.2464160.

10. Barrett E., Bolz-Tereick C. F., Killick R., Mount S., Tratt L. Virtual Machine Warmup Blows Hot and Cold // Proceedings of the ACM on Programming Languages. 2017. Vol. 1. OOPSLA. Art. 52. P. 1–27. DOI: 10.1145/3133876.

11. Richter S., Alvarez V., Dittrich J. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing // Proceedings of the VLDB Endowment. 2015. Vol. 9. No. 3. P. 96–107. DOI: 10.14778/2850583.2850585.

12. Herlihy M., Shavit N., Tzafrir M. Hopscotch Hashing // Distributed Computing – DISC 2008. LNCS 5218. Springer, 2008. P. 350–364. DOI: 10.1007/978-3-540-87779-0_24.

13. Ratanaworabhan P., Livshits B., Zorn B. G. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications // USENIX Conference on Web Application Development (WebApps ’10). 2010. P. 27–38.

14. V8 Team. How V8 Measures Real-World Performance // V8 Blog. 2016. December 21. URL: https://v8.dev/blog/real-world-performance

15. Schiavio F., Rosà A., Löff J., Bulej L., Tůma P., Binder W. MapReplay: Trace-Driven Benchmark Generation for Java HashMap // Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE ’26). 2026. P. 341–354. DOI: 10.1145/3777884.3797010.