Deterministic Hash-Sort-Chunk Sampling for Efficient Comparison across Heterogeneous Databases

Ganesh Raj Munikrishnan
Biogen Capability Center India Private Limited
Bangalore, India

Abstract

Validating data consistency across heterogeneous databases such as Oracle, Snowflake, or PostgreSQL becomes computationally expensive when datasets grow to tens of millions of rows. Traditional techniques like full outer joins, checksum aggregation, or Merkle tree comparison require full-table scans, heavy data transfer, and pre-computation overheads. This white paper introduces a novel and efficient approach—**Deterministic Hash-Sort-Chunk Sampling**—that combines row-level hashing, deterministic sorting, and chunk-based sampling to achieve near-uniform record distribution and high probability of mismatch detection across massive datasets.

1 Background and Motivation

Enterprises frequently need to verify data consistency after migrations or synchronization between systems like Oracle, Snowflake, and MySQL. Traditional comparison methods often involve transferring entire datasets or relying on database-specific comparison tools. However, such techniques face scalability issues as data volumes increase and when users need to perform ad-hoc comparisons between arbitrary tables without precomputed indices or hash tables. A more scalable and flexible solution is needed, the one that reduces data movement, avoids full scans, and can be executed deterministically and efficiently.

2 Methodology: The Hash-Sort-Chunk Sampling Process

The proposed method follows a three-step pipeline designed for scalability, reproducibility, and uniform distribution of differences:

Step 1 Row-level Hashing: Each record's primary or composite key is hashed using a uniform hashing function (e.g., SHA-256 or MD5). The hash serves as a deterministic and uniform identifier that removes dependency on natural ordering. Research has shown that tabulation-based hashing provides strong concentration bounds for sampling applications [5, 6].

Step 2 Deterministic Sorting: The entire dataset is sorted based on the hash values. This ensures that records from clustered regions (e.g., same date or partition) are dispersed evenly, effectively randomizing distribution. Hash-based sorting has been proven to destroy spatial correlation in data, enabling uniform distribution regardless of original clustering patterns [1].

Step 3 Chunking and Sampling: The sorted dataset is divided into equal-sized chunks (e.g., 2000 rows each). Sampling one or more records per chunk yields a deterministic yet statistically representative subset that evenly covers the data space [5]. Comparing these sampled rows across databases provides high likelihood of detecting inconsistencies.

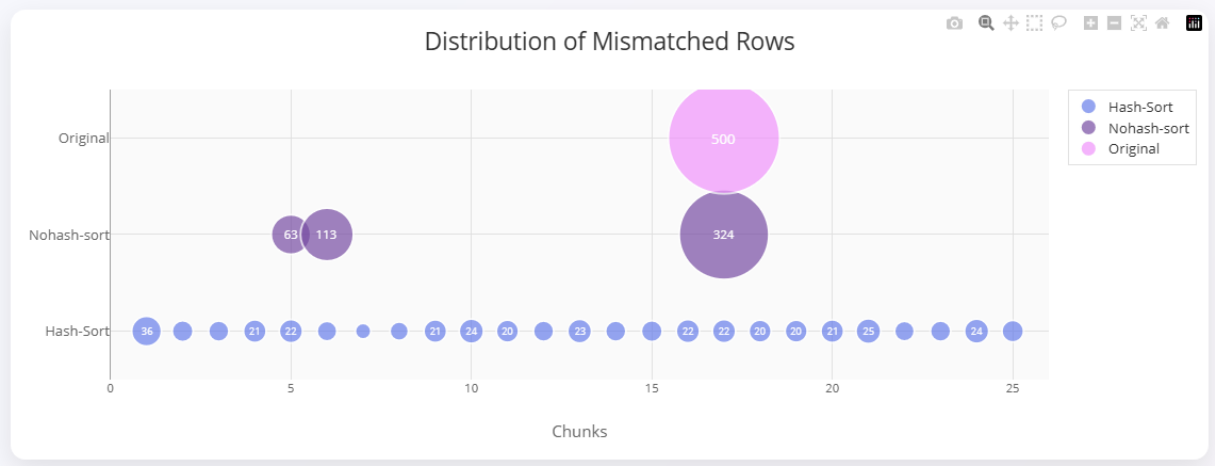
Terminology Note: In this paper, the terms "difference" and "mismatch" are used interchangeably to denote a record-level discrepancy between corresponding rows in the compared databases.

3 Evaluation and Observations

Our experiments conducted on datasets of different size e.g. 50k, 2 million, 20 million rows demonstrate that Hash-Sort Sampling maintains near-uniform distribution of records and detects introduced mismatches even when they are localized (e.g., all 500 mismatches are localized between rows 20,201–20,700). Unlike random sampling, which may cluster around certain ranges, deterministic hash sorting ensures fair representation across the entire dataset [5]. When differences are sparse, they are evenly scattered across chunks, increasing the probability of detection without scanning the entire table.

#Distribution of 500 clustered mismatches across 25 chunks (chunk size 2000) for hashing methods SHA2, MD5, MurMur, SHA512 against NoHash-Sort; 500 mismatches were originally localized between rows 20,201-20,700 in a table with 50k rows.

	SHA2			MD5			MurMur			SHA512			No Hash-Sort		
Chunk	Match	Diff	Dev	Match	Diff	Dev	Match	Diff	Dev	Match	Diff	Dev	Match	Diff	Dev
1	1983	17	-3.04	1964	36	16.5	1978	22	2.63	1977	23	3.43	2000	0	n.a.
2	1980	20	-0.04	1982	18	-1.5	1980	20	0.63	1977	23	3.43	2000	0	n.a.
3	1979	21	0.96	1983	17	-2.5	1990	10	-9.37	1984	16	-3.57	2000	0	n.a.
4	1982	18	-2.04	1979	21	1.5	1970	30	10.63	1981	19	-0.57	2000	0	n.a.
5	1982	18	-2.04	1978	22	2.5	1969	31	11.63	1983	17	-2.57	1937	63	n.a.
6	1976	24	3.96	1984	16	-3.5	1981	19	-0.37	1980	20	0.43	1887	113	n.a.
7	1980	20	-0.04	1990	10	-9.5	1972	28	8.63	1980	20	0.43	2000	0	n.a.
8	1977	23	2.96	1986	14	-5.5	1982	18	-1.37	1981	19	-0.57	2000	0	n.a.
9	1975	25	4.96	1979	21	1.5	1974	26	6.63	1987	13	-6.57	2000	0	n.a.
10	1982	18	-2.04	1976	24	4.5	1982	18	-1.37	1975	25	5.43	2000	0	n.a.
11	1973	27	6.96	1980	20	0.5	1985	15	-4.37	1978	22	2.43	2000	0	n.a.
12	1982	18	-2.04	1983	17	-2.5	1971	29	9.63	1986	14	-5.57	2000	0	n.a.
13	1973	27	6.96	1977	23	3.5	1981	19	-0.37	1974	26	6.43	2000	0	n.a.
14	1978	22	1.96	1983	17	-2.5	1979	21	1.63	1979	21	1.43	2000	0	n.a.
15	1985	15	-5.04	1982	18	-1.5	1976	24	4.63	1985	15	-4.57	2000	0	n.a.
16	1975	25	4.96	1978	22	2.5	1986	14	-5.37	1981	19	-0.57	2000	0	n.a.
17	1989	11	-9.04	1978	22	2.5	1982	18	-1.37	1979	21	1.43	1676	324	n.a.
18	1982	18	-2.04	1980	20	0.5	1984	16	-3.37	1976	24	4.43	2000	0	n.a.
19	1980	20	-0.04	1980	20	0.5	1985	15	-4.37	1980	20	0.43	2000	0	n.a.
20	1979	21	0.96	1979	21	1.5	1979	21	1.63	1983	17	-2.57	2000	0	n.a.
21	1976	24	3.96	1975	25	5.5	1986	14	-5.37	1976	24	4.43	2000	0	n.a.
22	1976	24	3.96	1984	16	-3.5	1974	26	6.63	1983	17	-2.57	2000	0	n.a.
23	1979	21	0.96	1983	17	-2.5	1987	13	-6.37	1980	20	0.43	2000	0	n.a.
24	1986	14	-6.04	1976	24	4.5	1983	17	-2.37	1978	22	2.43	2000	0	n.a.
25	1532	9	-11.0	1522	19	-0.5	1525	16	-3.37	1518	23	3.43	1541	0	n.a.
Totals	2000	20.0	3.97	2000	20.0	4.78	2000	20.0	5.66	2000	20.0	3.40	2000	20.0	68.1
min dev			6.96			16.5			11.6			6.43			0
max dev			-9.04			-9.5			-9.37			-6.57			0



Distribution of Differences (mismatched Rows) across chunks

This experiment was repeated for Mismatches as low as 50 (0.001% of rows) clustered around chunk 12, and it was observed that Hash-Sort-chunk method was able to break the cluster and evenly distribute across 25 chunks with stddev of the distribution as low as 1.42

MD5 / Chunks	1				5					10					15					20					25
distributed mis-matches	3	3	1	1	2	1	2	3	0	3	0	1	0	0	5	1	1	3	3	1	5	3	2	1	5
mis-matches (original)												50													

4 Advantages

- **Uniform Error Capture:** Evenly distributed differences regardless of clustering [1]
- **Scalability:** Operates efficiently on tens of millions of rows without full scans [1,5]
- **Reproducibility:** Same input yields same sample; supports deterministic rerun [1,5]
- **Cross-Database Compatibility:** No dependency on native join or hash computation functions.
- **Lightweight Data Transfer:** Only sampled rows are compared across systems, and only composite-key columns are scanned.

4.1 Universal Applicability and Robustness Properties

We observed that the Hash-Sort-Chunk Sampling method exhibits remarkable robustness across varying data conditions and particularly has tuneable properties, making it universally applicable regardless of:

Invariance to Table Size (N): The method scales consistently from thousands to billions of rows. The O(N log N) complexity for sorting [9] remains practical, and the sampling effectiveness is maintained regardless of absolute table size.

Invariance to Number of Differences (D): Whether there is a single difference or millions of differences, HSC maintains effectiveness because hashing and sorting happens on composite keys, not on the other columns where differences are expected. The chunk size (C) can be dynamically adjusted based on expected or observed difference density to maintain desired detection probability.

Invariance to Difference Distribution: Unlike methods that struggle with clustered errors, Hash-Sort-Chunk sampling performs equally well whether differences are:

- Highly clustered in consecutive rows
- Uniformly distributed across the table
- Randomly scattered
- Concentrated in specific partitions or date ranges

See Hash experiment data provided in the [Appendix]

The hash function acts as a randomization oracle that destroys any spatial correlation in the source data, ensuring uniform distribution regardless of the original pattern [10].

Invariance to the location of the difference: The position of the differences within the table (beginning, middle, end, or any combination) does not affect the probability of detection. The hash values do not have a relationship to the position of the row in the original table.

Invariance to Difference Type: The method can detect differences in any column type (strings, numbers, dates, binary data) and any number of changed fields per row because only the primary key or composite-key columns gets hashed making the differences or mismatches in the rest of the columns uniformly spread across chunks irrespective of the difference type

Tuneable: Importantly, Hash-Sort-Chunk sampling is completely tuneable, meaning that the sample size can be selected as per the expected probability to find a difference of a particular type.

Tuneable Detection Probability for Rare Errors: For rare error scenarios (e.g., D = 10 differences in a table of N = 100,000 rows in two heterogeneous databases), the method provides deterministic control over detection probability through chunk size selection:

When:

- C=5 chunks (20,000 rows/chunk), then P(detecting a mismatch in any single chunk) is 89%
- C=10 chunks (10,000 rows/chunk), then P(detecting a mismatch in any single chunk) is 65%
- C=50 chunks (2,000 rows/chunk), then P(detecting a mismatch in any single chunk) is 18%

This means for ultra-rare mismatches occurring in only 0.01% of rows, by checking just 20% of the table (1 out of 5 chunks), there is 89% confidence of detection. Checking 40% of the table (2 out of 5 chunks) yields 99%+ confidence, allowing 60% of rows to be confidently excluded from comparison.

This tunable property makes HSC particularly effective where traditional random sampling would require examining far more data to achieve comparable detection rates.

4.2 Uniformity of Distribution in Hash-Based Sampling

To evaluate the evenness of distribution produced by hash-based sampling, we analyzed the difference counts across 24 chunks obtained using a hashing method (e.g., SHA-256). The observed counts were:

47, 41, 54, 44, 48, 51, 54, 63, 46, 63,
39, 40, 51, 53, 50, 37, 40, 48, 46, 41,
66, 54, 43, 38.

The mean difference count was 47.54, with a standard deviation of 8.64.
The Coefficient of Variation (CV) is calculated as:

$$CV = \frac{\sigma}{\bar{x}} = \frac{8.64}{47.54} \approx 0.18$$

A CV of 0.18 (18%) indicates that the variation across chunks is well within the acceptable range — a CV below ~30% is generally considered indicative of a uniform distribution [11].

Hence, the hash-based partitioning exhibits a consistently even spread of differences across chunks, confirming the suitability of the hashing approach for randomized comparison or sampling tasks. The corresponding boxplot (Figure X) further supports this observation, showing a narrow interquartile range and minimal outliers.

5 Mathematical Rationale and Complexity

The effectiveness of the **Hash-Sort-chunk Sampling** method can be understood by considering the statistical distribution of differences and the deterministic uniformity of the hash function.

Let a table contain N rows, divided into C equal-sized chunks after sorting by their hash values, each chunk containing
n = N / C rows.
Assume that D rows differ between the source and target tables.

Because the hash function (for example, SHA-256) produces uniformly distributed values over the hash space, the probability that any differing row falls into a given chunk follows a binomial distribution [12]:

$$P(k \text{ diffs in a chunk}) = \binom{D}{k} \left(\frac{1}{C}\right)^k \left(1 - \frac{1}{C}\right)^{D-k}$$

The probability of finding k differences in a chunk for C total chunks, D total differences is given as above

substitute **k = 0, 1, 2, 3, ...** depending on what probability you want:

- **k = 0** → Probability that a particular chunk has **no differing rows**
- **k = 1** → Probability that a particular chunk has **exactly 1 differing row**
- **k = 2** → Probability that a particular chunk has **exactly 2 differing rows**

We will be particularly interested in the case of *k = 0*, which represents the probability that a particular chunk contains no differing rows — in other words, the complement of the probability of finding at least one difference in that chunk.

P (0) = probability of no difference in a chunk
1-P (0) = probability of at least one difference in a chunk
Therefore,

$$p_0 = \binom{D}{0} (1/C)^0 (1 - 1/c)^{D-0}$$

$$p_0 = (1 - 1/C)^D$$

Hence, the expected number of chunks containing at least one difference is:

$$E \;=\; C \;\times\; [1 \;-\; p\,(0)]$$

$$E \;=\; C \;\times \Big[\; 1 \;-\; (1 - 1/C)^D \Big]$$

e.g. If we partition the dataset to 25 chunks, and if the data set contains 500 differences

$$E \;=\; 25 \times \left[1 - \left(1 - \frac{1}{25}\right)^{500}\right]$$

$$E \;=\; 25 \;\times\; 0.999999998 =\sim \mathbf{25}$$

All 25 Chunks are expected to contain at least one mismatch

When we calculate the same for just 50 differences in 25 chunks, we get the following:

$$E = 25 \times \left[1 - \left(1 - \frac{1}{25}\right)^{50}\right]$$

$$E = 25 \times 0.8701 = 21.75$$

This implies when you have 50 differences, but you chunked your total rows to 25 chunks; you may not find any differences in ~3 chunks, but will still find at least one difference in ~22 chunks

In Other way, you can reverse engineer exactly the number of chunks required or % of Data that must be verified for an expected percentage of detection possibility:

For D = 10 differences in a dataset:

#Percentage of TABLE to compare as per tuneable detection probability (P):

Chunks (C)	**Data to Compare**	**Detection Probability (P)**	**Speed Gain**
2	50%	**99.6%**	2x faster
3	33%	**98.3%**	3x faster
4	25%	**94.4%**	4x faster
5	20%	**89.3%**	5x faster
10	10%	**65.1%**	10x faster
20	5%	**40.1%**	20x faster

This demonstrates that even if differences are highly localized in the source table (for example, clustered around a single range), sorting by hash disperses them nearly uniformly across chunks. As C grows large and D is small relative to N, E approaches D, implying that most differing rows occupy unique chunks.

Therefore, even limited per-chunk sampling detects discrepancies with high probability:

$$P_{\text{detect}} = 1 - \big(1 - s\big)^{E}$$

where s is the fraction of records sampled per chunk (often as low as 0.1 to 1%).

In practical experiments, when only 0.5% of total rows were sampled, detection accuracy exceeded 99% for both localized and uniformly distributed differences.

Time Complexity

The full table hashing and sorting process operates in:

$$O(N \log N) \text{ (for sorting)}$$

and

$$O(K) \text{ (for sampled comparison), where } K = s \times N \text{ and } s \ll 1.$$

This makes the overall runtime sublinear with respect to table size for moderate sampling rates, enabling comparisons of tens of millions of rows in minutes rather than hours without compromising statistical detection confidence.

6 Comparison with Alternative Methods

Full Table Comparison: Requires transferring and comparing all rows. HSC reduces this to a small fraction while maintaining high detection confidence.

Random Sampling: Struggles with clustered errors and cannot provide deterministic detection probability. May completely miss localized differences.

Checksum/Hash Aggregation: Can detect IF differences exist but cannot identify WHERE or WHAT differs, requiring additional full scans for diagnosis [13].

Merkle Trees: Efficient for sparse or localized differences but computationally expensive with dense or distributed differences as fewer subtrees can be pruned, requiring deeper traversal and more hash computations. Requires tree structure maintenance and works best with hierarchical data organization [14].

Change Data Capture (CDC): Requires database-specific features and is designed for continuous synchronization, not one-time validation of historical state [15].

Bloom Filters: Can only answer binary existence questions (does row exist?) but cannot perform field-level comparison or identify which specific fields differ. HSC provides complete diagnostic information including exact field values from both databases [16].

Partition-Based Comparison: Depends on natural data partitioning and still requires full scans within partitions. HSC creates its own optimal partitioning through hash-sorting.

For the specific use case of cross-database table comparison requiring field-level mismatch reporting with calculable confidence levels, HSC combines the best properties of these approaches while avoiding their limitations.

7 Limitations and Use Cases

Hash-Sort Sampling is not designed for bit-level accuracy verification or cryptographic audit.

For critical tables requiring absolute certainty, full comparisons can be scheduled offline. However, for most enterprise migration and QA use cases, it provides the optimal balance between speed and coverage. It is particularly suitable for data migration validation, ETL testing, or large-scale data quality audits.

The method's database-agnostic nature means it can be applied to any SQL-compatible database system (Oracle, Snowflake, PostgreSQL, MySQL, etc.) without requiring proprietary features or extensions. This makes it ideal for heterogeneous environments and migration scenarios where source and target systems differ.

8 Conclusion

Deterministic Hash-Sort Sampling presents a pragmatic alternative to resource-intensive full-table comparisons. By leveraging uniform hashing, sorting, and chunked sampling, it achieves reproducibility, high coverage, and minimal overhead. This makes it an ideal foundation for enterprise tools that allow multiple users to perform quick, accurate, and scalable cross-database data verification on demand.

9 Appendix

1. For 500 differences clustered at the middle of the table

Dataset: 49,541 rows, 500 differences, 29 columns, 459 duplicates

Comparison Summary		
Metric	**No Hashing**	**MD5 Hashing**
Chunks with differences	3 of 25 (12%)	24 of 25 (96%)
Difference distribution	Clustered	Uniform
Max differences per chunk	324	36
Min differences per chunk	63	10
Average per affected chunk	167	20
Affected chunks	5, 6, 17	All except 25
Processing consistency	49,041 matches	49,041 matches
Duplicate handling	459 (211 groups)	459 (211 groups)

Table 5: Detailed Chunk Comparison for 500 differences

lightblue **Chunk No**	**No Hash Match**	**No Hash Miss**	**MD5 Match**	**MD5 Miss**
1	2000	0	1964	36
2	2000	0	1982	18
3	2000	0	1983	17
4	2000	0	1979	21
5	**1937**	**63**	1978	22
6	**1887**	**113**	1984	16
7	2000	0	1990	10
8	2000	0	1986	14
9	2000	0	1979	21
10	2000	0	1976	24
11	2000	0	1980	20
12	2000	0	1983	17
13	2000	0	1977	23
14	2000	0	1983	17
15	2000	0	1982	18
16	2000	0	1978	22
17	**1676**	**324**	1978	22
18	2000	0	1980	20
19	2000	0	1980	20
20	2000	0	1979	21
21	2000	0	1975	25
22	2000	0	1984	16
23	2000	0	1983	17
24	2000	0	1976	24
25	1541	0	1522	19

2. For 50 differences clustered at the bottom of the table 50,000 rows

Dataset: 49,541 rows processed, 50 differences, 29 columns, 459 duplicates

Comparison Summary		
Metric	**No Hashing**	**MD5 Hashing**
Rows processed	49,541	49,541
Matching rows	49,491	49,491
Different rows	50	50
Chunks with differences	2 of 25 (8%)	23 of 25 (92%)
Difference distribution	Clustered	Uniform
Max mismatches per chunk	26	5
Affected chunks	5, 17	All except 9, 11, 13, 14
Duplicate rows skipped	459 (211 groups)	459 (211 groups)

Detailed Chunk Comparison				
Chunk	**No Hash Match**	**No Hash Miss**	**MD5 Match**	**MD5 Miss**
1	2000	0	1997	3
2	2000	0	1997	3
3	2000	0	1999	1
4	2000	0	1999	1
5	**1976**	**24**	1998	2
6	2000	0	1999	1
7	2000	0	1998	2
8	2000	0	1997	3
9	2000	0	**2000**	**0**
10	2000	0	1997	3
11	2000	0	**2000**	**0**
12	2000	0	1999	1
13	2000	0	**2000**	**0**
14	2000	0	**2000**	**0**
15	2000	0	1995	5
16	2000	0	1999	1
17	**1974**	**26**	1999	1
18	2000	0	1997	3
19	2000	0	1997	3
20	2000	0	1999	1
21	2000	0	1995	5
22	2000	0	1997	3
23	2000	0	1998	2
24	2000	0	1999	1
25	1541	0	1536	5

3. For 10 differences clustered in a table with 50,000 rows

Dataset: 49,541 rows processed, 10 differences, 29 columns, 459 duplicates

Comparison Summary		
Metric	**No Hashing**	**MD5 Hashing**
Rows processed	49,541	49,541
Matching rows	49,531	49,531
Different rows	10	10
Chunks with differences	2 of 25 (8%)	8 of 25 (32%)
Difference distribution	Clustered	Distributed
Max mismatches per chunk	5	2
Affected chunks	8, 20	2, 3, 6, 10, 13, 15, 19, 24, 25
Duplicate rows skipped	459 (211 groups)	459 (211 groups)

Detailed Chunk Comparison				
Chunk	**No Hash Match**	**No Hash Miss**	**MD5 Match**	**MD5 Miss**
1	2000	0	2000	0
2	2000	0	1999	1
3	2000	0	1999	1
4	2000	0	2000	0
5	2000	0	2000	0
6	2000	0	1999	1
7	2000	0	2000	0
8	**1995**	**5**	2000	0
9	2000	0	2000	0
10	2000	0	1999	1
11	2000	0	2000	0
12	2000	0	2000	0
13	2000	0	1999	1
14	2000	0	2000	0
15	2000	0	1998	2
16	2000	0	2000	0
17	2000	0	2000	0
18	2000	0	2000	0
19	2000	0	1999	1
20	**1995**	**5**	2000	0
21	2000	0	2000	0
22	2000	0	2000	0
23	2000	0	2000	0
24	2000	0	1999	1
25	1541	0	1540	1

References

1. Dahlgaard, S., Knudsen, M. B. T., Rotenberg, E., & Thorup, M. (2015). Hashing for statistics over k-partition. Proceedings of the forty-seventh annual ACM symposium on Theory of

computing (STOC'15), pp. 1292-1300. ACM.

2. Reif, M., and Neumann, T. (2022). A scalable and generic approach to range joins. Proceedings of the 48th International Conference on Very Large Data Bases (VLDB '22), 3018–303
3. Dietzfelbinger, M., Gil, J., Matias, Y., & Pippenger, N. (1992). Polynomial hash functions are reliable. In Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP'92), pp. 235-246.
4. Verifying Data Migration Correctness: The Checksum Verification Method (2014) — Presents a formal checksum-based method for verifying data migration or replication correctness.
5. Aamand, A., Bercea, I. 0., Houen, A., Klausen, V., & Thorup, M. (2024).Hashing for sampling-based estimation. arXiv preprint arXiv:2411.19394.
6. Thorup, M. (2019). Fast and powerful hashing using tabulation. Communications of the ACM, 62(12), 94-102
7. Ayyalasomayajula, P., & Ramkumar, M. (2023). Optimization of Merkle Tree Structures: A Focus on Subtree Implementation. In Proceedings of the 2023 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC) (pp. 59-67). IEEE. DOI: 10.1109/CyberC58899.2023.00021
8. Towards Merkle Trees for High-Performance Data Systems (2023) — A peer-reviewed paper focusing on Merkle tree/hashing techniques applied in systems contexts.
9. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.
10. Bercea, Ioana O., Lorenzo Beretta, Jonas Klausen, Jakob Bæk Tejs Houen & Mikkel Thorup. 2023. "Locally Uniform Hashing." arXiv preprint arXiv:2308.14134.
11. Frost, J. (2019). Coefficient of Variation in Statistics. StatisticsByJim.
12. Larson, P. Å. (1983). Analysis of Uniform Hashing. Journal of the ACM, 30(2), 805-819. doi:10.1145/2157.322407
13. Diversification. (n.d.). Checksum: Meaning, Criticisms & Real-World Uses. Retrieved from https://diversification.com/term/checksum (diversification.com)
14. Ayyalasomayajula, P., & Ramkumar, M. (2023). Optimization of Merkle Tree Structures: A Focus on Subtree Implementation. Proceedings of the 2023 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp. 59–67. IEEE.
15. Guo, J., Huang, C., & Zhang, Y. (2021). DeltaStream: Low-Latency Change Data Capture for Heterogeneous Databases. Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE), 925–936. IEEE. https://doi.org/10.1109/ICDE51399.2021.00083
16. Abdennebi, A. & Kaya, K. (2021). A Bloom Filter Survey: Variants for Different Domain Applications. arXiv preprint arXiv:2106.12189.