CLOUDEXIFY

PROJECT #: 230601

Instructor: Dr. Sarel Cohen

OUR TEAM

Ori
Braverman

Shachar
Levy

Maayan
Mashhadi

THE RESEARCHERS

Dr. Sarel Cohen

Moshik Hershkovitch

Senior Research Scientist @ IBM Research
Ph.D. Candidate @ TAU

Daniel Cohen

M.Sc. Student
Supervised by Dr. Sarel Cohen & Moshik Hershkovitch

Yarin Pinyan

Director of Security Research @ Startup

THE PROBLEM

- Lack of efficient secondary indexing for unsorted data.
- Existing indexing methods are inadequate for handling unsorted data effectively.
- Difficulty in retrieving specific information from unsorted datasets.
- Absence of a straightforward solution for organizing and accessing unsorted data efficiently.

TARGET & INTENTION

01 Address the challenge of efficient secondary indexing for unsorted data.

02 To improve query performance, increased flexibility, and reduced data storage overhead in DynamoDB tables, by creating secondary index using LSI.

03 Extend a novel approach that enhances the retrieval of specific information from unsorted datasets.

04 Colaborate with IBM research and startups for industrial aplicativity.

05 Improve the overall performance and effectiveness of secondary indexing methods.

SOLUTION

Permutation Vector

For mapping predictions into the underlying unsorted base data.

Learned Index

Which maps a lookup key to a bounded search range.

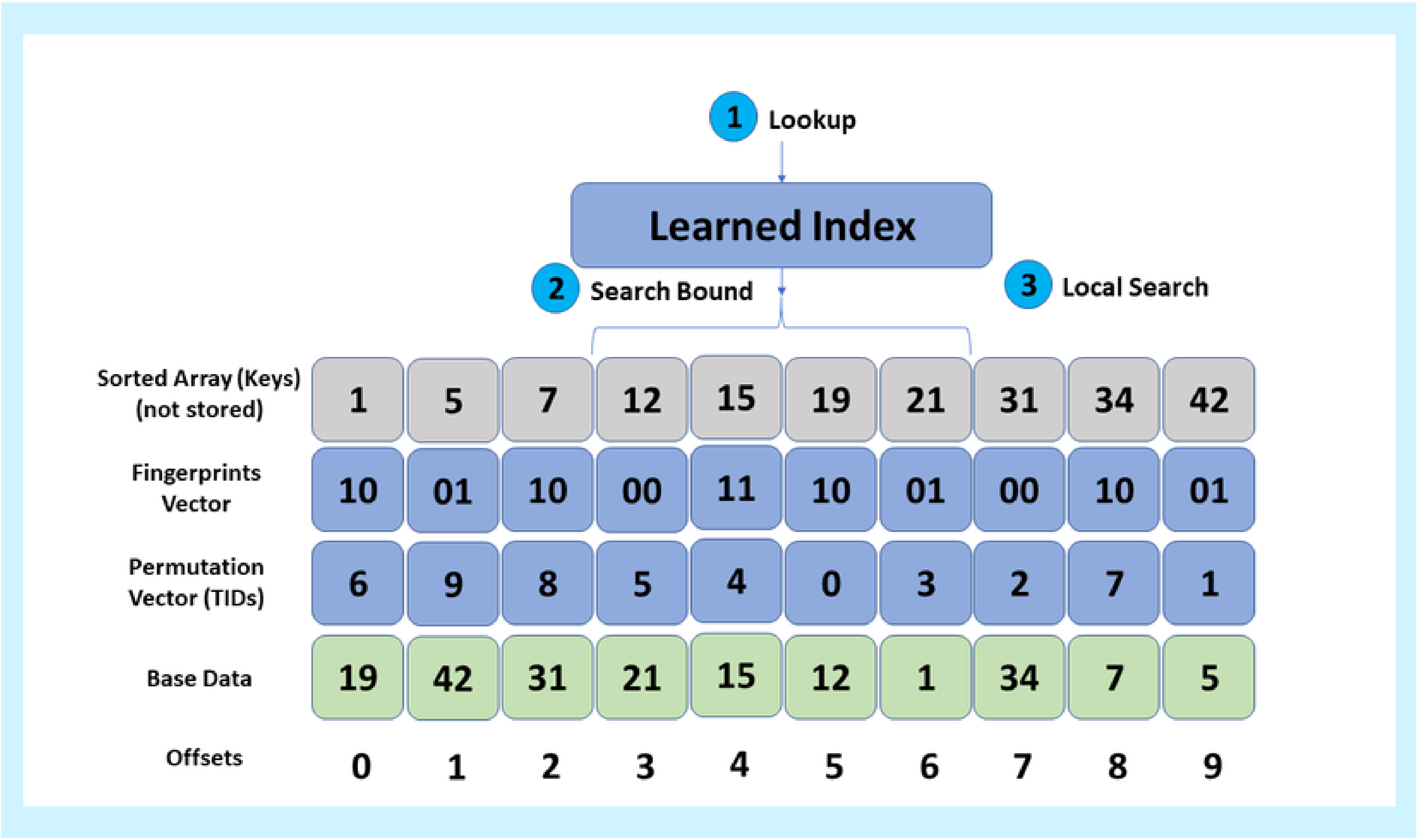
Fingerprint Vector

To prune unnecessary accesses during equality lookups.

CLOUDEXIFY

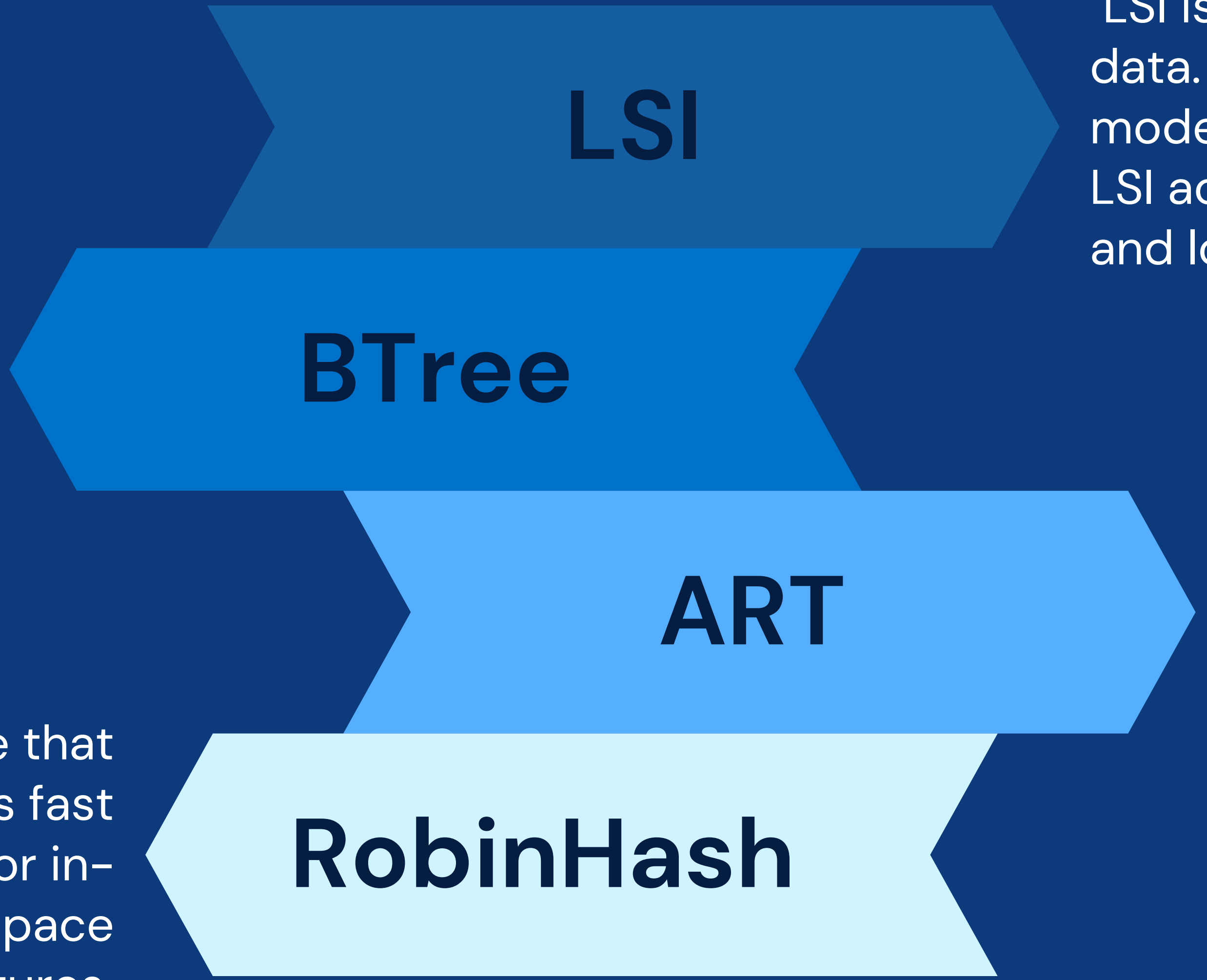
ARCHITECTURE AND THE PROCESS OF LSI

The local search uses the permutation vector to locate corresponding entries in the base data.
The fingerprint vector prunes unnecessary accesses.



Sorted Array (Keys) (not stored)	1	5	7	12	15	19	21	31	34	42
Fingerprints Vector	10	01	10	00	11	10	01	00	10	01
Permutation Vector (TIDs)	6	9	8	5	4	0	3	2	7	1
Base Data	19	42	31	21	15	12	1	34	7	5
Offsets	0	1	2	3	4	5	6	7	8	9

OTHER SOLUTIONS AND BENCHMARKS



LSI is a learned index structure that can index unsorted data. It uses machine learning techniques to build a model that approximates the distribution of the data. LSI achieves a good trade-off between space efficiency and lookup performance.

BTree is a balanced tree-based index structure that organizes data in a sorted order. It is commonly used in traditional database systems. BTree supports efficient insertion, deletion, and search operations. It is suitable for both disk-based and memory-based systems.

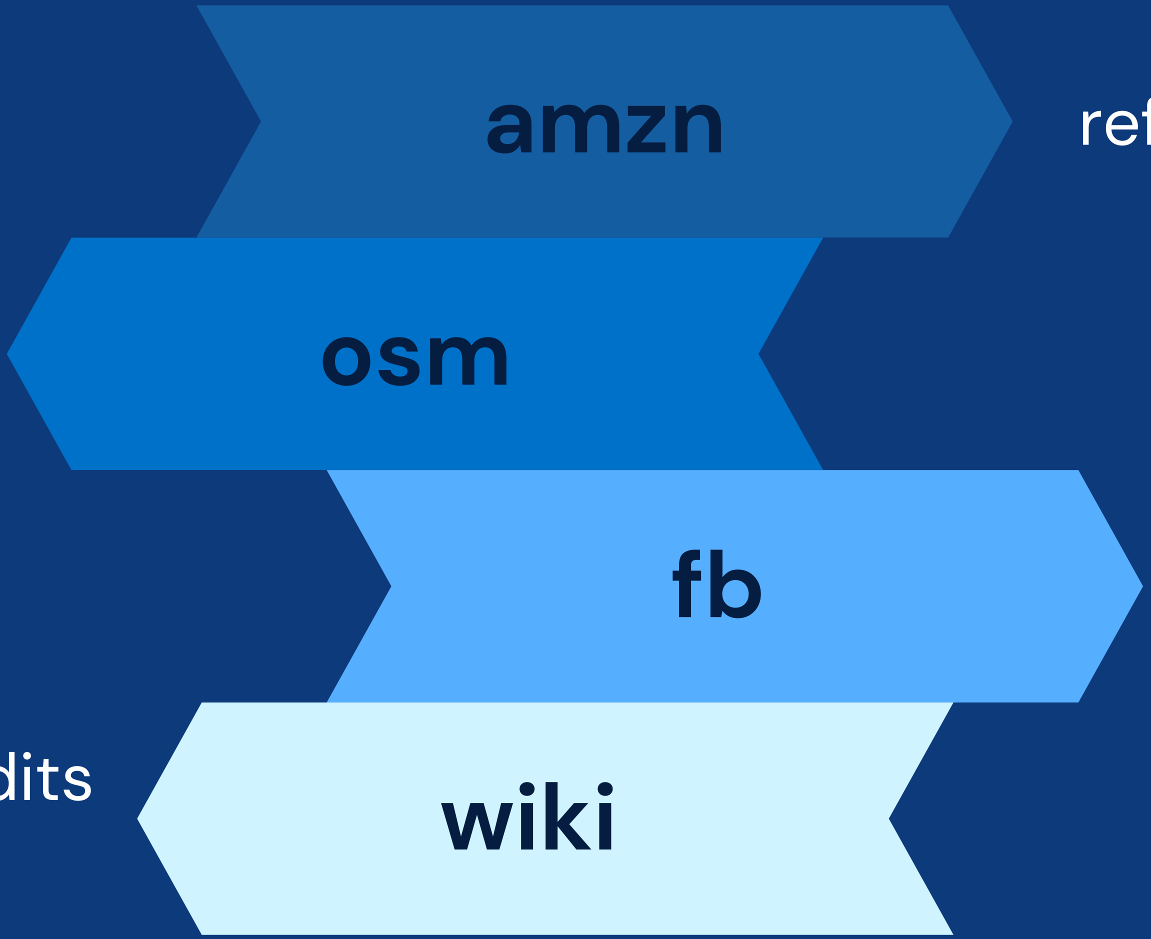
The Adaptive Radix Tree (ART) is a data structure designed for efficient indexing and retrieval of key-value pairs in databases. It adapts its structure dynamically based on the data distribution, resulting in efficient search operations. ART is known for its low memory overhead and high performance.

RobinHash is a hash table-based index structure that uses a robin-hood hashing algorithm. It provides fast lookup and insertion operations, making it suitable for in-memory databases. RobinHash consumes more space compared to other index structures.

compare their performance in terms of build times, lookup latency, and space usage.

Comparison Between Other Solutions

Datasets used to evaluate LSI:



amzn refers to book popularity data

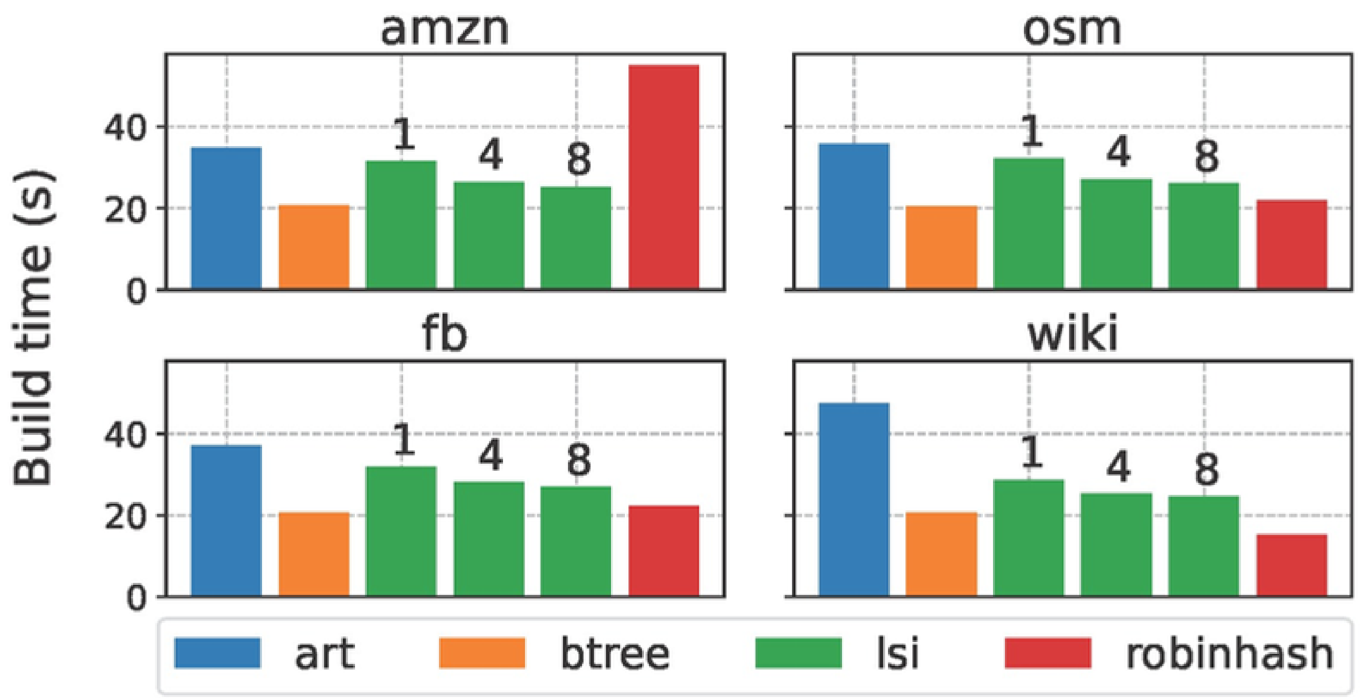
osm to cell IDs from Open Street Map

fb refers to randomly sampled Facebook user IDs

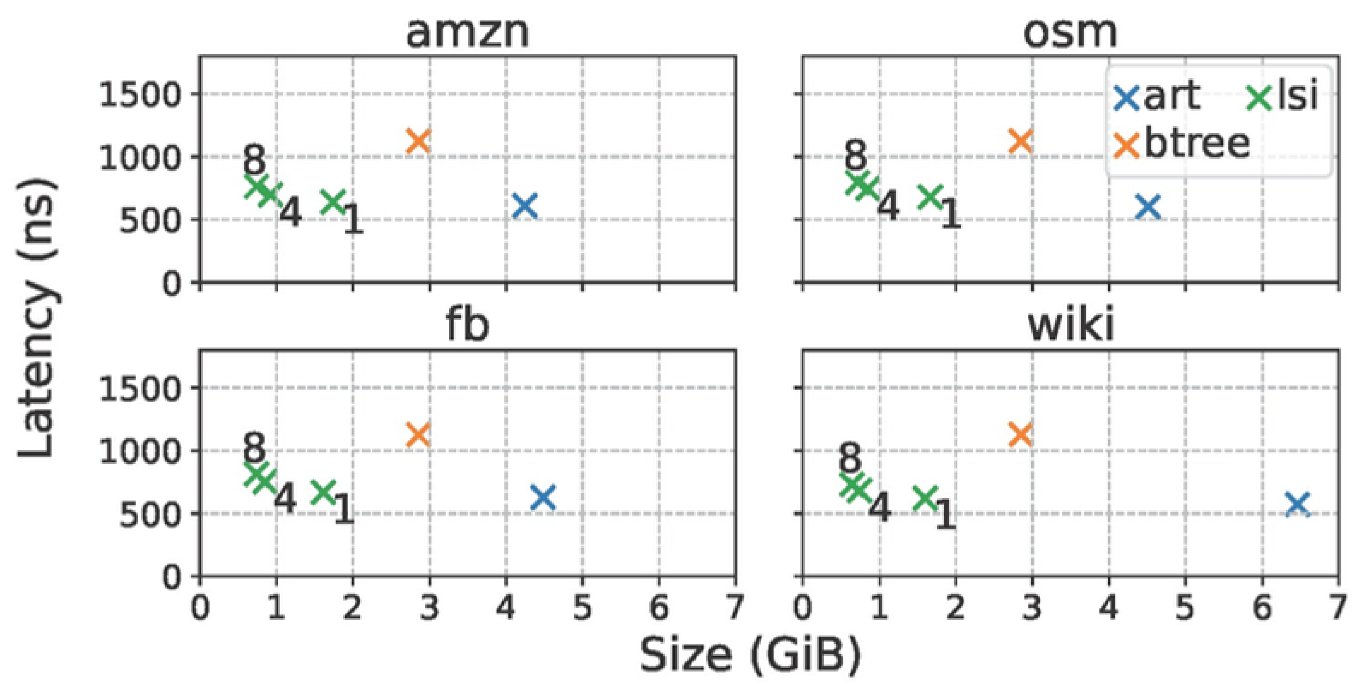
wiki refers to timestamps of edits from Wikipedia.

COMPARISON BETWEEN OTHER SOLUTIONS

Build times in seconds for different index structures.
The BTree index structure achieves the lowest build times, followed by RobinHash and LSI with an error bound of 8. ART, on the other hand, has the highest build times.

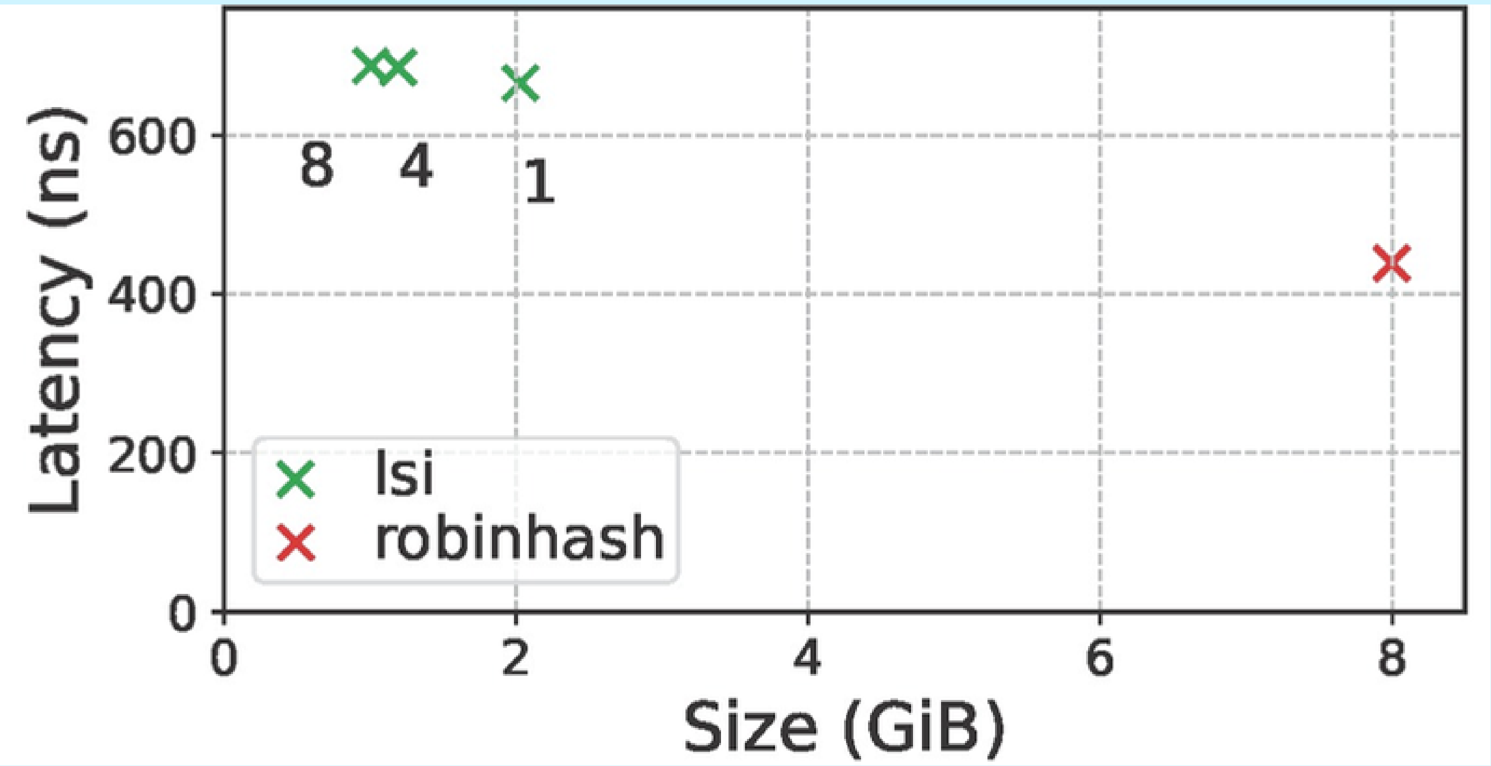


lower-bound lookups using non-existing keys. The graph represents the latency (in nanoseconds) of the lookups, and the text annotations on the graph indicate the error bounds.



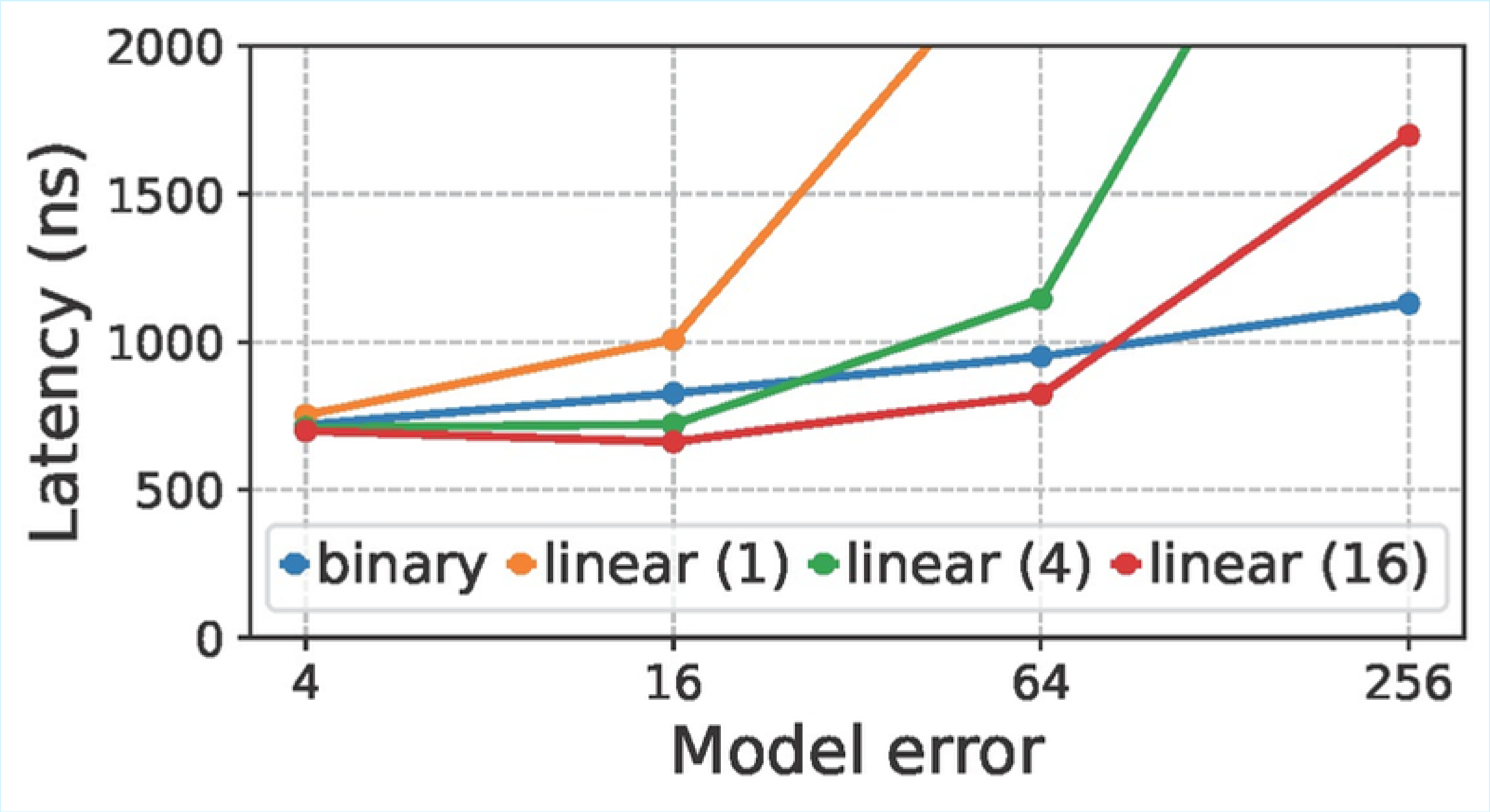
COMPARISON BETWEEN OTHER SOLUTIONS

Equality lookups on the amzn dataset comparing LSI to RobinHash.
RobinHash has a bit less latency then LSI's. However, RobinHash also consumes 4 times the amount of space compared to LSI.

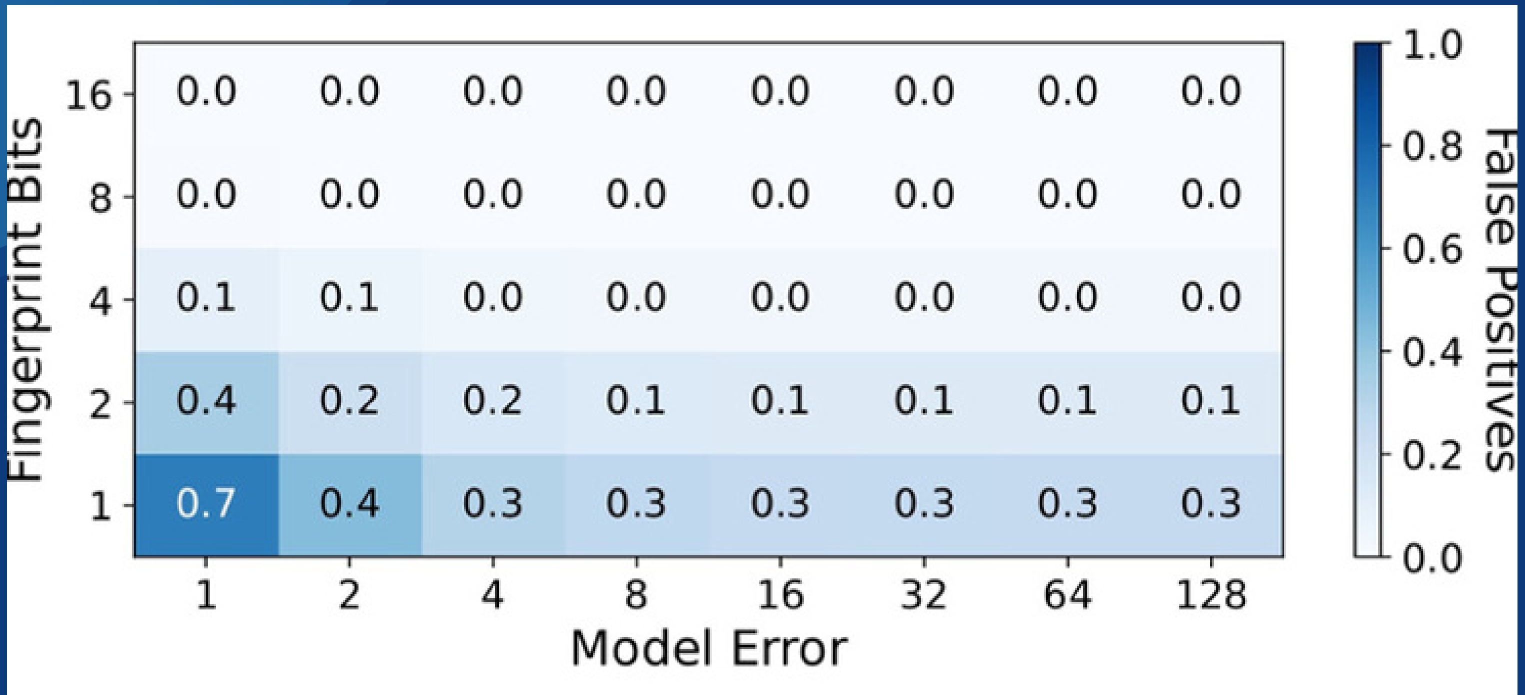


COMPARISON BETWEEN SEARCH METHODS

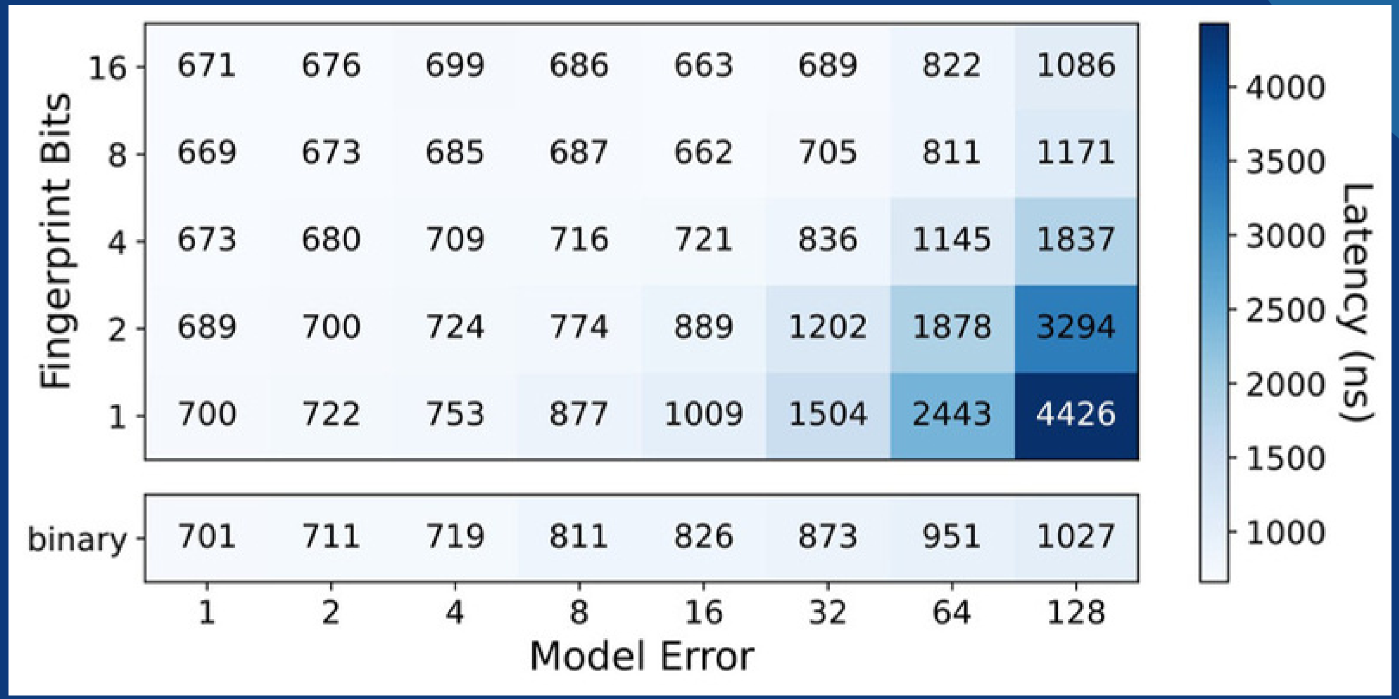
Compares binary search and linear search with different fingerprint sizes. The brackets in the figure indicate the number of fingerprint bits used for each search method.



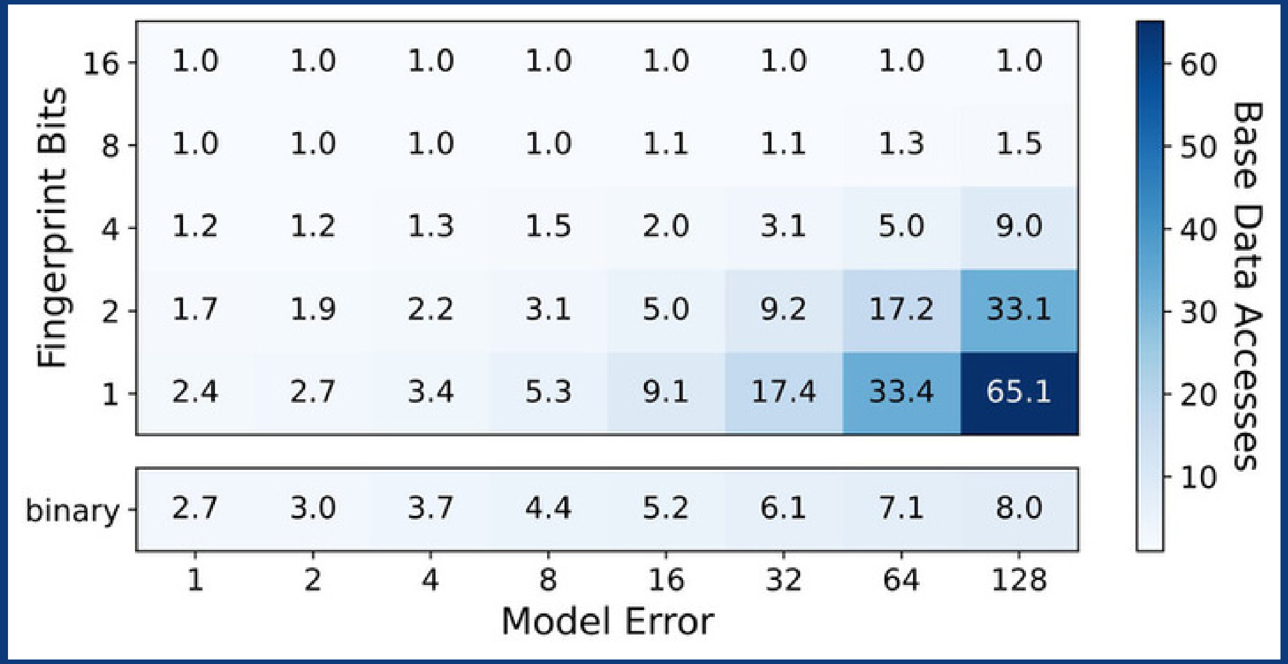
ANALYSIS OF FINGERPRINT VECTOR



Fingerprint Bits	1	2	4	8	16	32	64	128
16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0
2	0.4	0.2	0.2	0.1	0.1	0.1	0.1	0.1
1	0.7	0.4	0.3	0.3	0.3	0.3	0.3	0.3

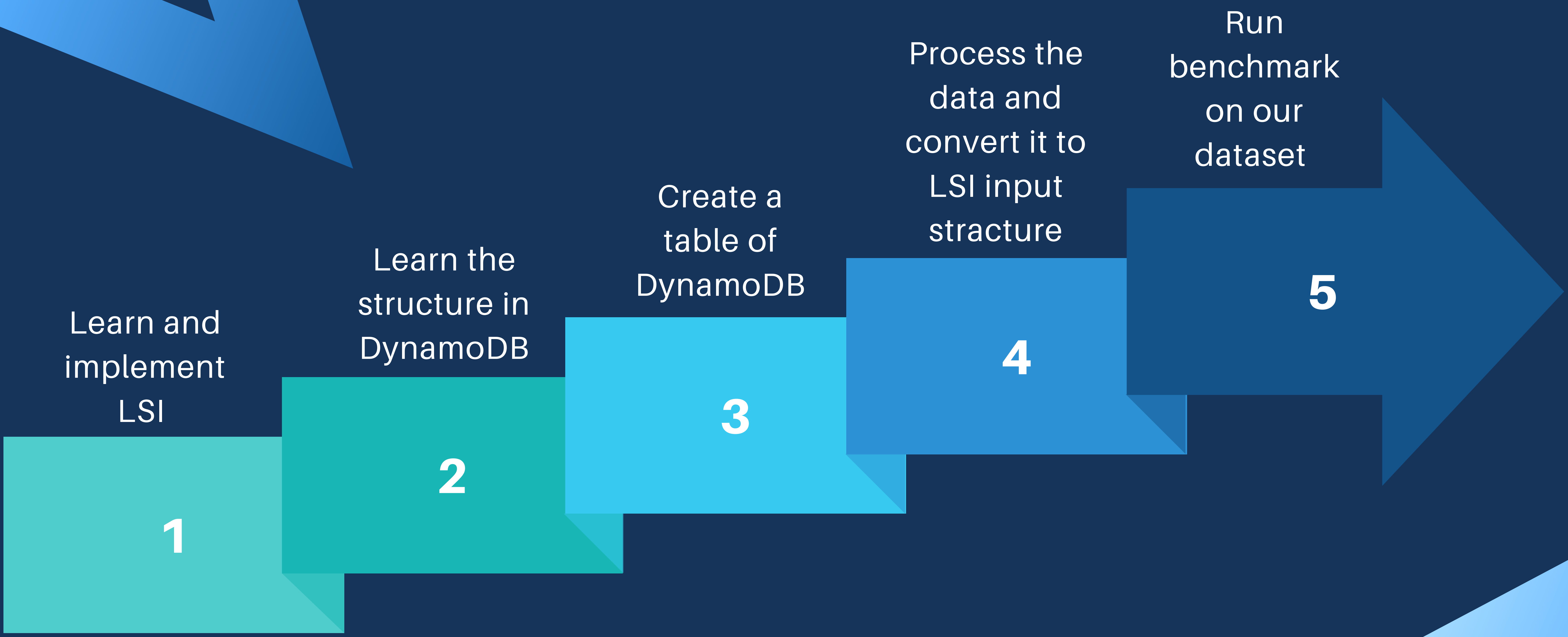


Fingerprint Bits	1	2	4	8	16	32	64	128
16	671	676	699	686	663	689	822	1086
8	669	673	685	687	662	705	811	1171
4	673	680	709	716	721	836	1145	1837
2	689	700	724	774	889	1202	1878	3294
1	700	722	753	877	1009	1504	2443	4426
binary	701	711	719	811	826	873	951	1027



Fingerprint Bits	1	2	4	8	16	32	64	128
16	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
8	1.0	1.0	1.0	1.0	1.1	1.1	1.3	1.5
4	1.2	1.2	1.3	1.5	2.0	3.1	5.0	9.0
2	1.7	1.9	2.2	3.1	5.0	9.2	17.2	33.1
1	2.4	2.7	3.4	5.3	9.1	17.4	33.4	65.1
binary	2.7	3.0	3.7	4.4	5.2	6.1	7.1	8.0

OUR WORK PROCESS

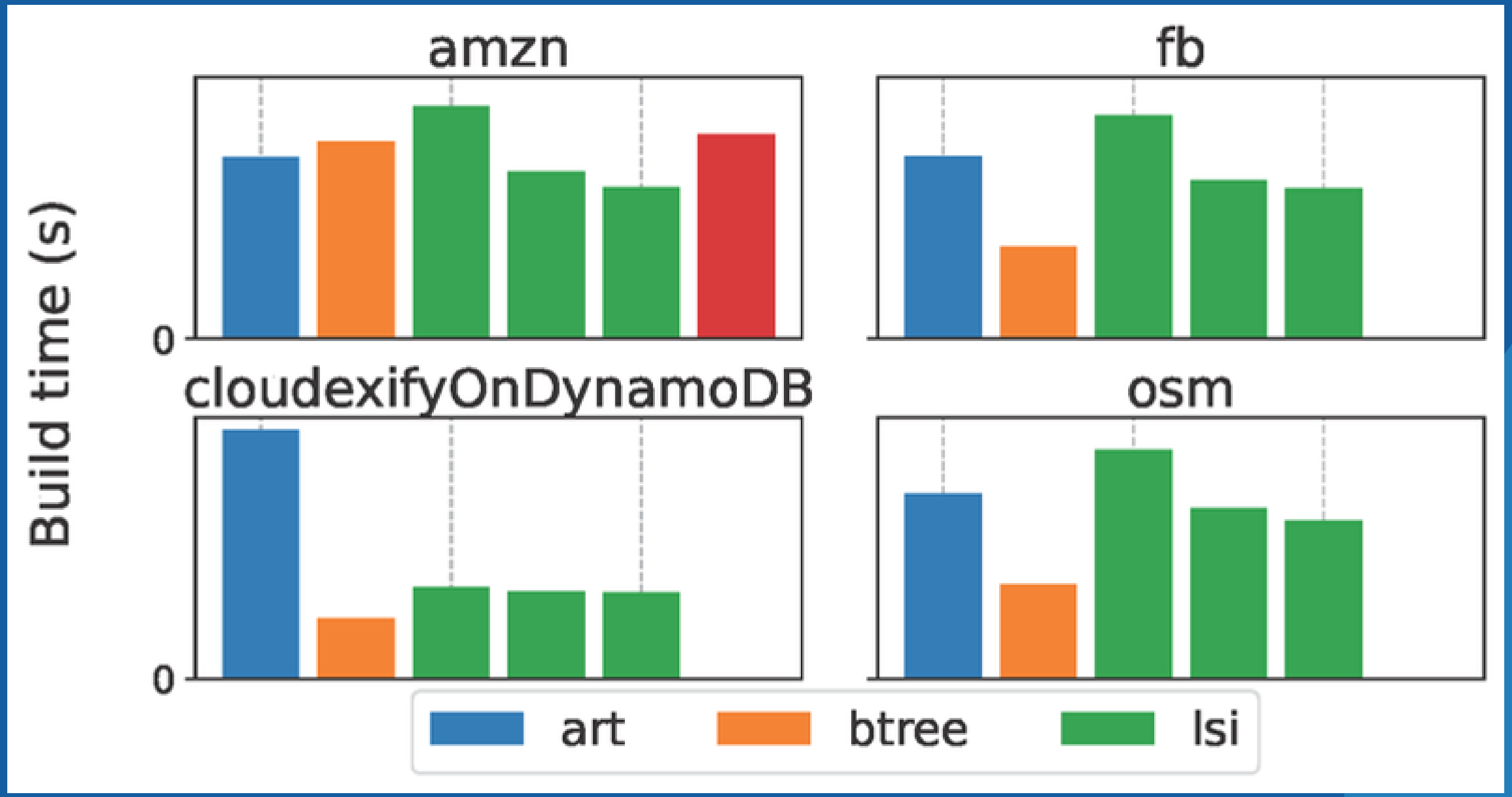


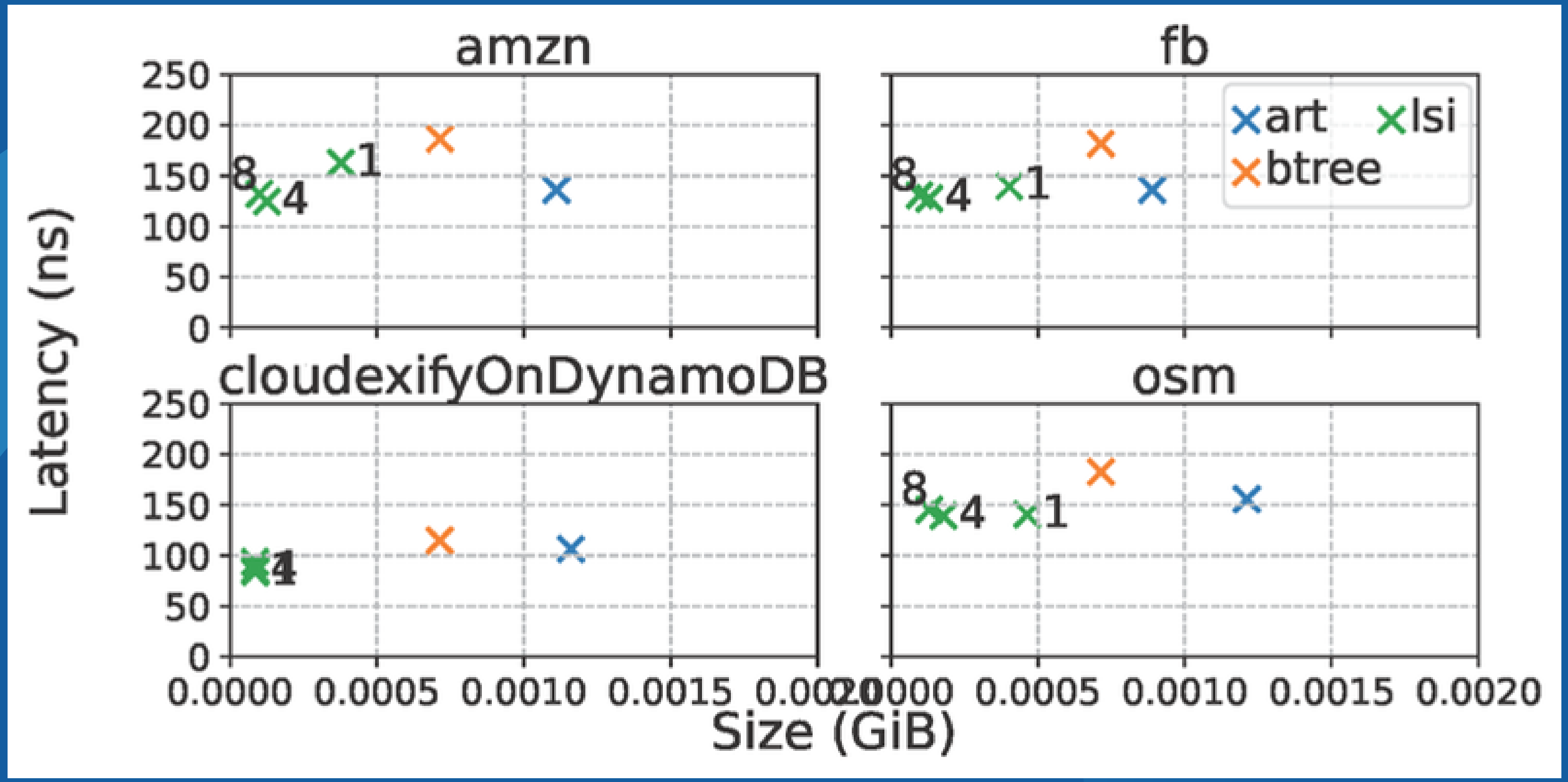
1. Learn and implement LSI
2. Learn the structure in DynamoDB
3. Create a table of DynamoDB
4. Process the data and convert it to LSI input structure
5. Run benchmark on our dataset

DYNAMO-DB

- Amazon DynamoDB is a fully managed NoSQL database service offered by Amazon Web Services (AWS), designed to provide high availability, scalability, and low-latency access to data.
- Its underlying structure is based on a key-value store, where each item in a table is uniquely identified by a primary key.
- DynamoDB supports secondary indexes, which enhance query flexibility.
- These secondary indexes use a B-tree data structure for efficient indexing, making it possible to retrieve data quickly based on attributes other than the primary key.

DynamoDB Benchmark results





המכללה האקדמית תל־אביב־יפו

Thank's For Watching

Connect with us.

https://github.com/shacharlevy/Cloudexify