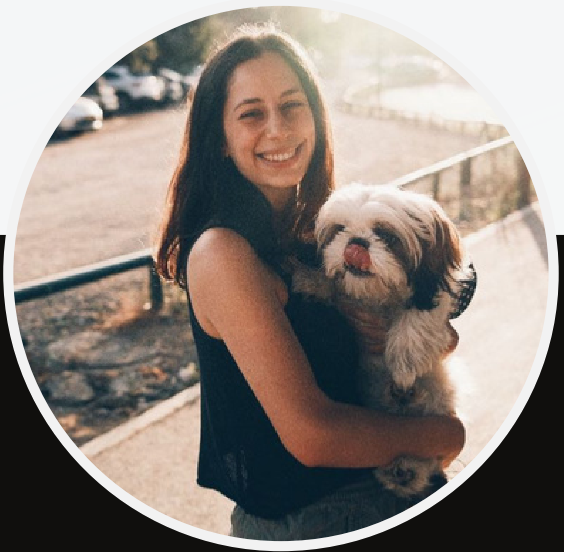
BLINDI PRESENTS NVRAM

OUR TEAM

Yoav
Shwartz

Ranan
Harpak



Lir
Mimrod

Tal
Meschiany

Mentor: Dr. Sarel Cohen
Project No. 231306

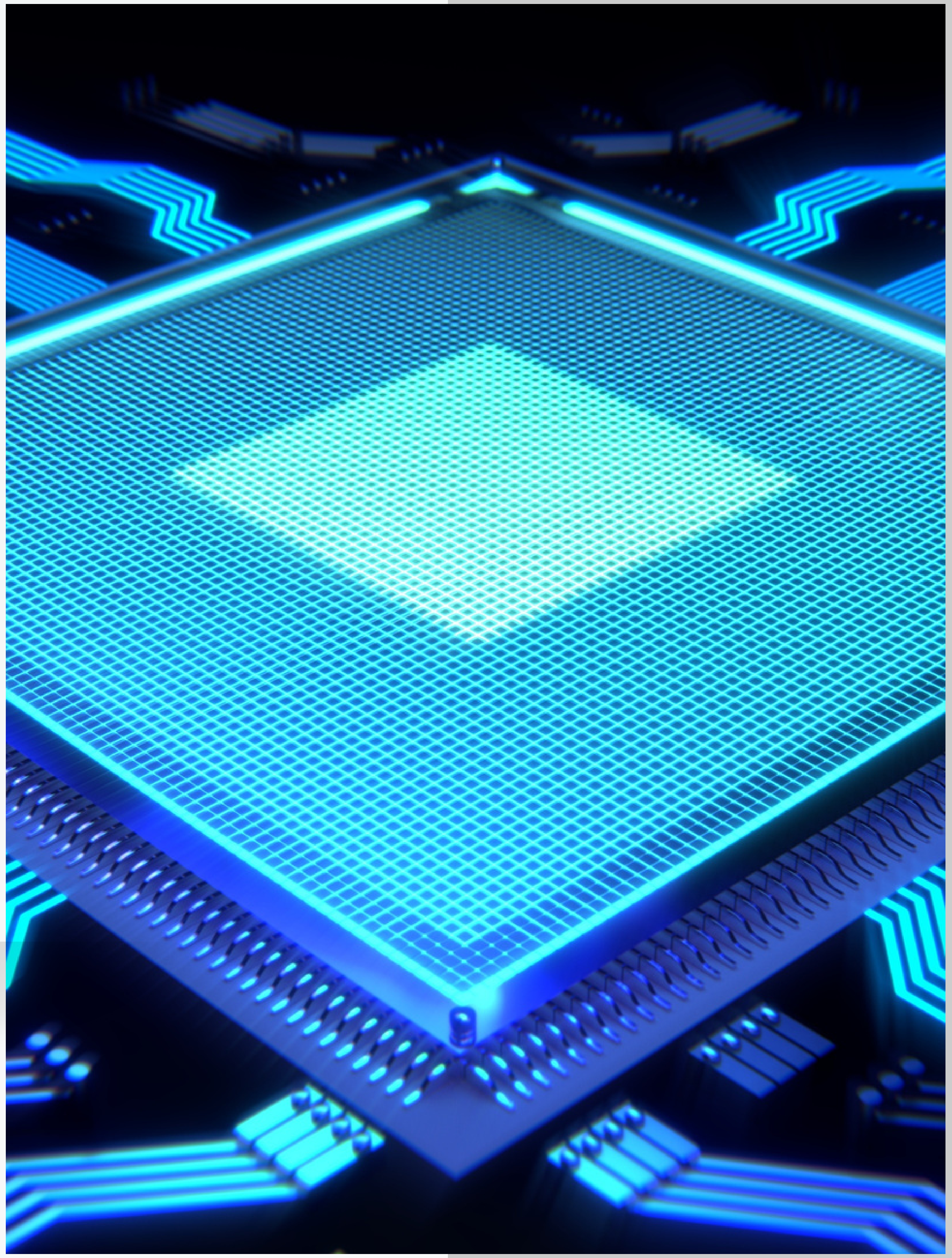
THE PROBLEM

- In-memory database management systems (DBMSs) are important elements of data pipelines
- A major source of DBMS memory overhead—which stresses memory provisioning demands—is storage of indexed keys by the DBMS index data structures
- Lack of efficient secondary indexing for unsorted data
- Difficulty in retrieving specific information from unsorted data
- Absence of a straightforward solution for organizing and accessing unsorted data efficiently
- Crashes could lead to data loss

SOLUTION

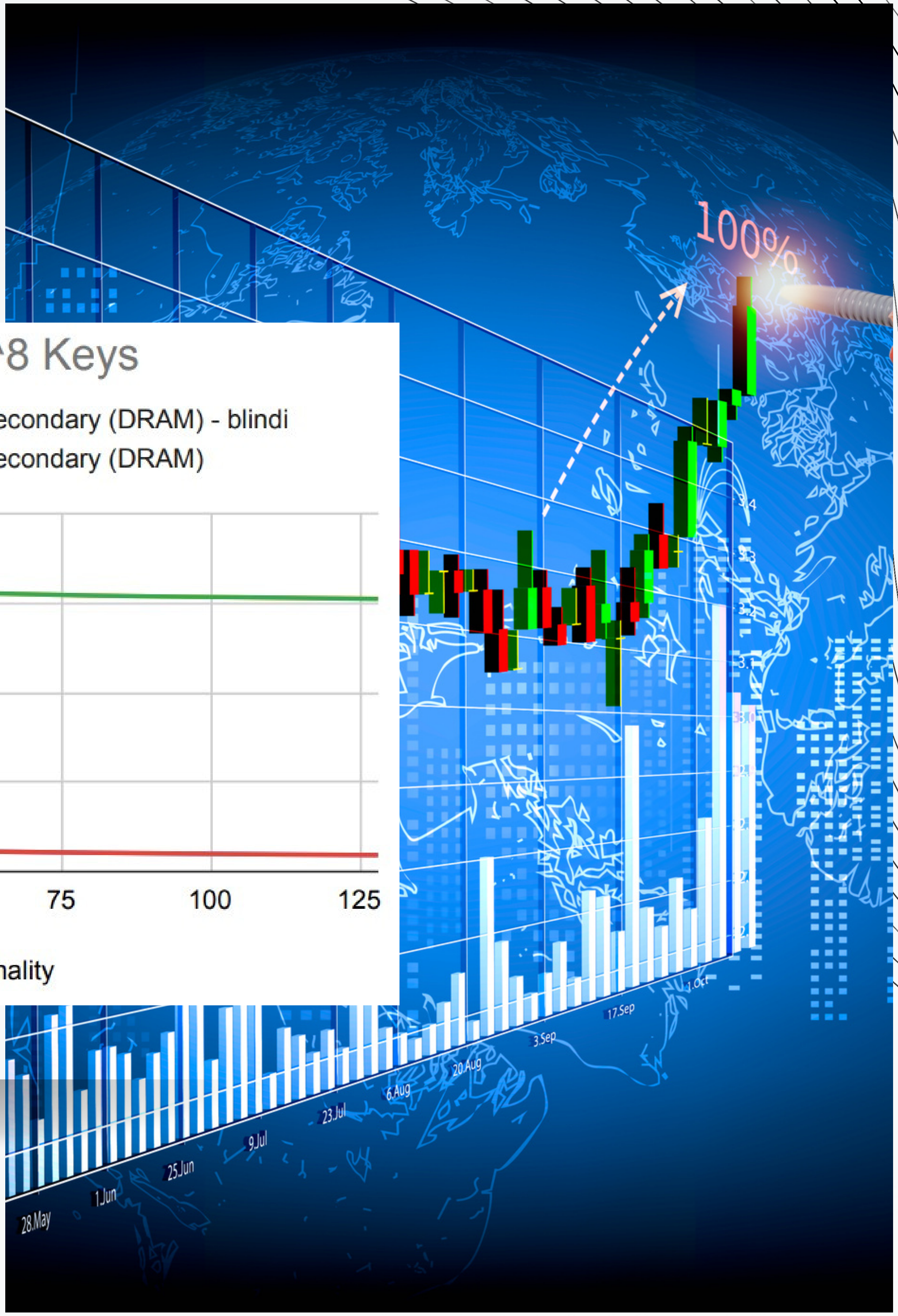
- compact data structure with data in NVRAM and a tree index in DRAM
- The index may be quickly reconstructed from the data after crash

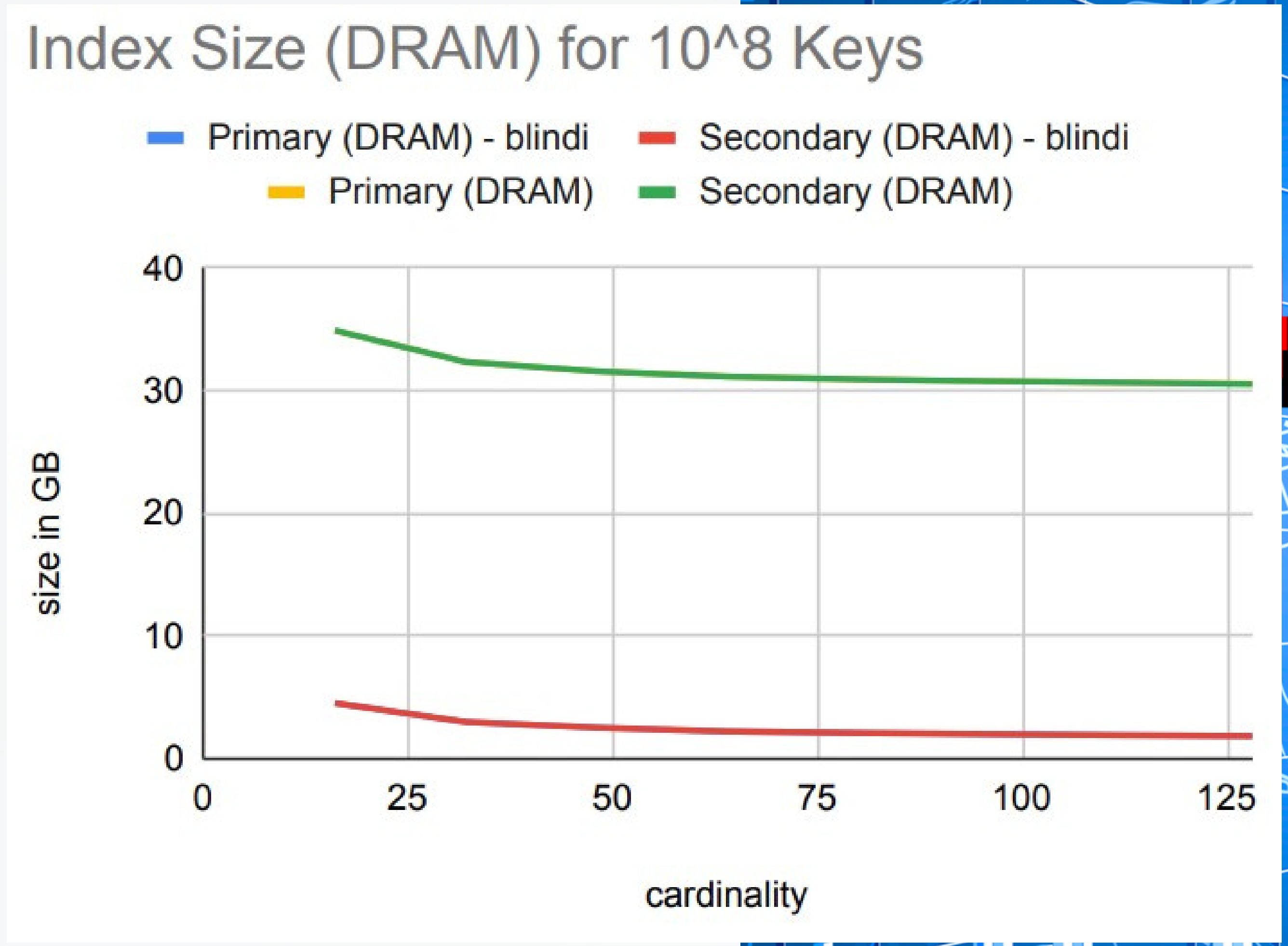
- NVRAM has high capacities and significant scalability compared to DRAM
- A standard NVRAM can store up to 6TB of memory compared to 64GB of a standard DRAM



BENCHMARKS

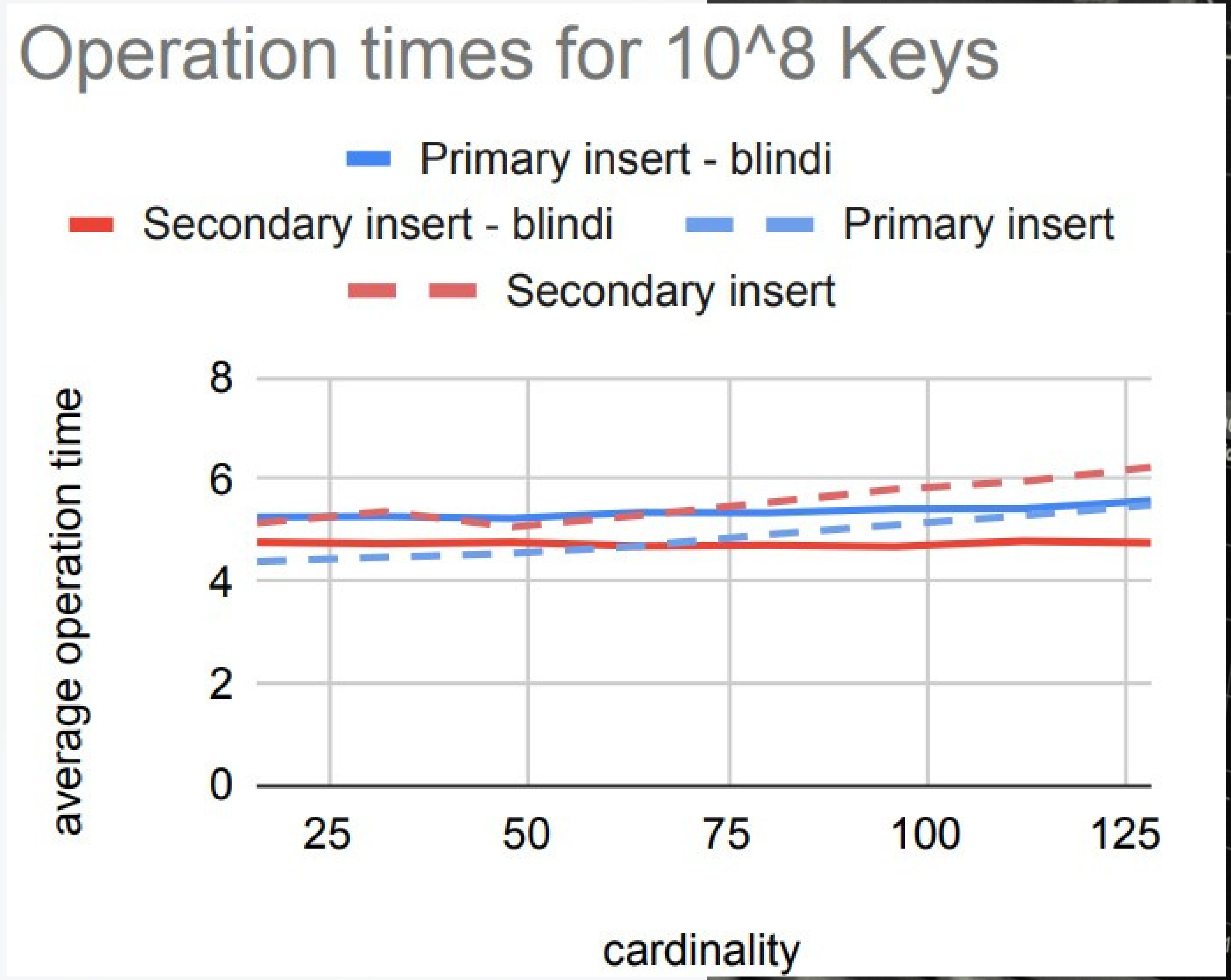
- we compare what happens for various leaf cardinalities, i.e., the amount of keys that are stored in a leaf of the tree
- Our structure improves DRAM usage by roughly an order of magnitude. The cardinality does not have an impact on insert times for our structure, so they do not serve as an obstacle to increase leaf sizes





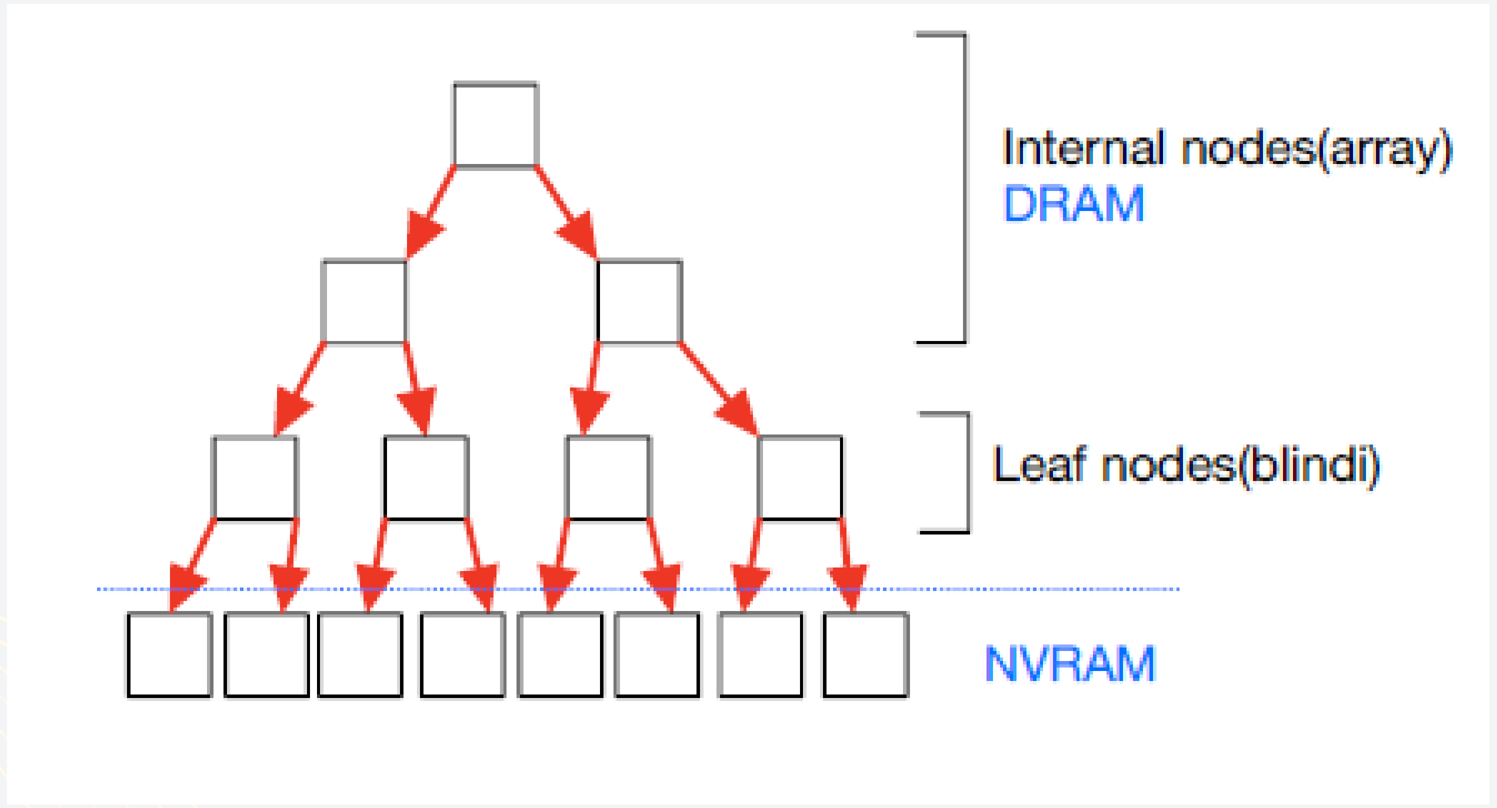
BENCHMARKS

- we compare what happens for various leaf cardinalities, i.e., the amount of keys that are stored in a leaf of the tree.
- Our structure improves DRAM usage by roughly an order of magnitude. The cardinality does not have an impact on insert times for our structure, so they do not serve as an obstacle to increase leaf sizes



ARCHITECTURE

We base our data structure on the µTree. The aim of the µTree is to provide a fast data index, which can quickly be restored from NVRAM in case of a crash or interruption. While the space usage within NVRAM is already quite small, there is much to be gained for the tree structure in the DRAM. Especially for larger keys, the space used in DRAM is quite high, as all the keys are duplicated in both DRAM and NVRAM. We aim to remove this duplication by using blind tries in the DRAM instead of storing the keys. Whereas the leaf nodes of the original µTree use the same array to point to the data, we use a more sophisticated approach: Within the node we store a blind trie to map from keys to a pointer to the data.



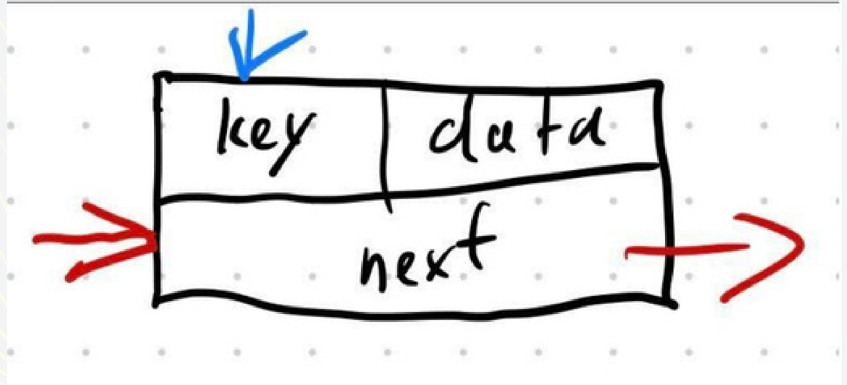
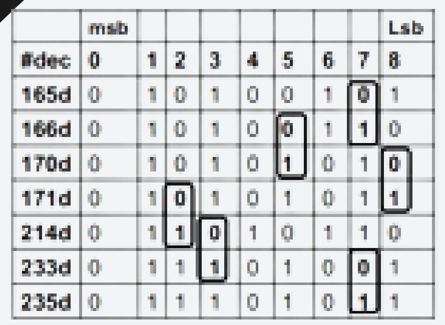


Fig. 2. A node of the linked list in NVRAM

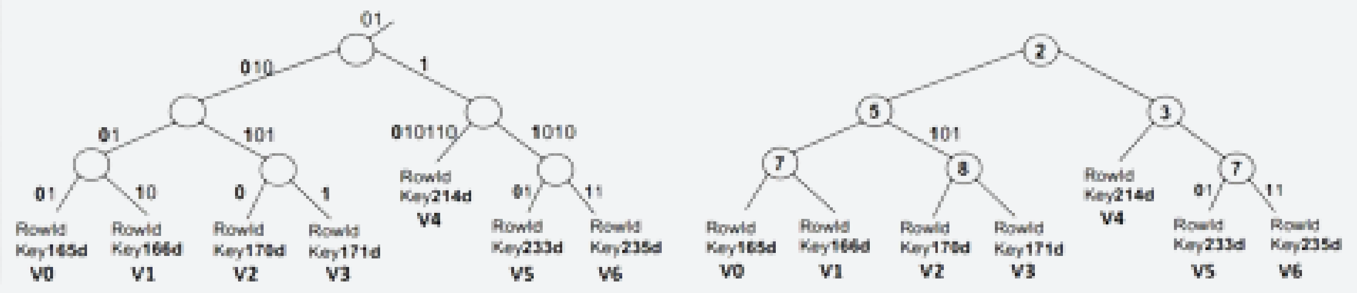
ARCHITECTURE

A blind trie is a compressed trie that does not label the edges but instead records in each node the position of the bit in which the node's children differ. The blind trie does not explicitly store all key bits, and so is more compact than a compressed trie



	msb								Lsb
#dec	0	1	2	3	4	5	6	7	8
165d	0	1	0	1	0	0	1	0	1
166d	0	1	0	1	0	0	1	1	0
170d	0	1	0	1	0	1	0	1	0
171d	0	1	0	1	0	1	0	1	1
214d	0	1	1	0	1	0	1	1	0
233d	0	1	1	1	0	1	0	0	1
235d	0	1	1	1	0	1	0	1	1

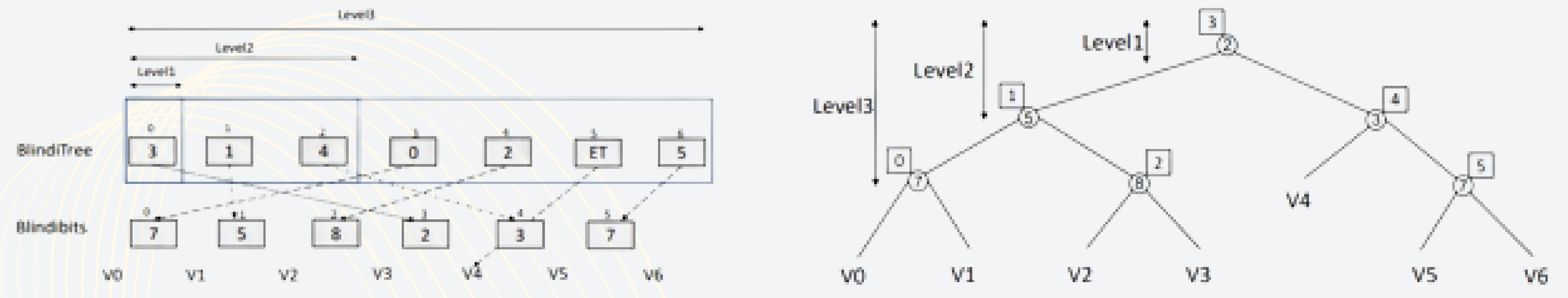
(a) Indexed keys and their binary representation.



(b) Compact trie.

(c) Blind trie.

Figure 2: Different tries indexing a 7-row table whose rows are indexes by keys shown in (a). Rectangles mark the first bit in which two consecutive keys differ. The notation Vx refers to the value associated with the x-th key, e.g., a tuple identifer.



(a) SeqTree arrays. *BlindiTree* is the physical layout of the tree shown in (b). *BlindiBits* is the trie's SeqTrie representation. Numbers above array entries indicate their index.

(b) SeqTree logical representation. The boxed number next to a node shows its index in the *BlindiBits* array.

Figure 3: SeqTree representation for the key set of Figure 2a with tree of height 3.

CONCLUSION

In our project we implemented code to be used with NVRAM which is a relatively new hardware.

With the increase of the amount of data stored, a more memory efficient way to store the required keys.