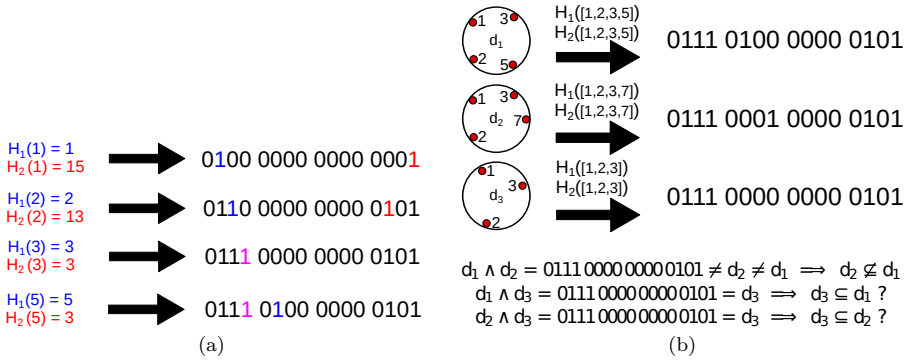


Fig. 4. (a) Process of generating binary signatures. (b) Subset checking using signatures.

the set. Each hash function maps an object identifier to a position inside the signature (bucket) and sets a number of 1-bit according to the given identifier. Figure 4(a) shows the step-by-step signature generation for the set $\{1, 2, 3, 5\}$ highlighting the bits set by each hash function regarding every object identifier. In the figure, $H_1$ and $H_2$ refer, respectively, to the SpookyHash[5] and MurMurHash[6], which are fast hashes implementations and with good non-linearity (measured by avalanche criterion[7]). Note that some collisions may happen between the hash functions (bits represented in purple in Figure 4(a)). Therefore, the ideal size of the signatures depends on the cardinality of the sets. However, we recommend to be up to 64 bits, since we have run tests (using a 64-bit architecture, which is the currently the "standard") that showed that **AND** bitwise operations of signatures greater than this size may suffer a performance drop.

Figure 4(b) illustrates the process of filtering through binary signatures. The idea is to avoid performing a set intersection operation to determine if a disk is subset/superset of another disk in case this is surely false. In order to achieve this, we apply an **AND** bitwise operation between the two signatures from the disks. If the result of the operation is equal to one of the operands, and then this operand may be a subset of the other. Otherwise, we can surely say that no disk is a superset of the other. In Figure 4(b), the set intersection between $d_1$ and $d_2$ is avoided as none of these sets is a subset of the other according to the signature checking. However, as this approach is subject to false positives, it is necessary to perform a set intersection operation as a post-processing step regarding $d_1, d_3$ and $d_2, d_3$. Nevertheless, this step should eliminate many false-negatives depending on the chosen hash functions. The primitives of performing subset queries using binary signatures are described in details in [Goel and Gupta 2010].

### 4.3 Inverted Index-based Join Between Sets of Consecutive Timestamps

After finding all disks for a given time instant, we then need to join them with others from the previous timestamp. A straightforward way to join disks is to process all disks from one timestamp with the other timestamp, and then check for the joining condition (i.e., if two disks have at least $\mu$ objects in common). However, this process is computationally costly since we have to perform, for each timestamp, set intersection operations for all candidates of a timestamp with all the maintained flocks of the previous one. Instead, here we propose the use of inverted indexes to speed up the step of joining disks across two consecutive timestamps.

Inverted index is a well-known method employed to index documents and then efficiently search for terms in the index [Zobel and Moffat 2006]. Usually, an inverted index has a list of keys that are the

[5] `burtleburtle.net/bob/hash/spooky.html`
[6] Murmurhash 2.0: `sites.google.com/site/murmurhash`
[7] `floodyberry.com/noncryptohashzoo`