Optimization techniques

Bloom filters

Bloom filters

- Probabilistic data structure, check membership for a value in a set.

- How it works: S, set of n values → *const* * *n* bits
  calculate hash(v) ∈ $[1, const * n]$
  set bit hash(v) to 1

  Test w ∈ S → h(w) = 1 ?

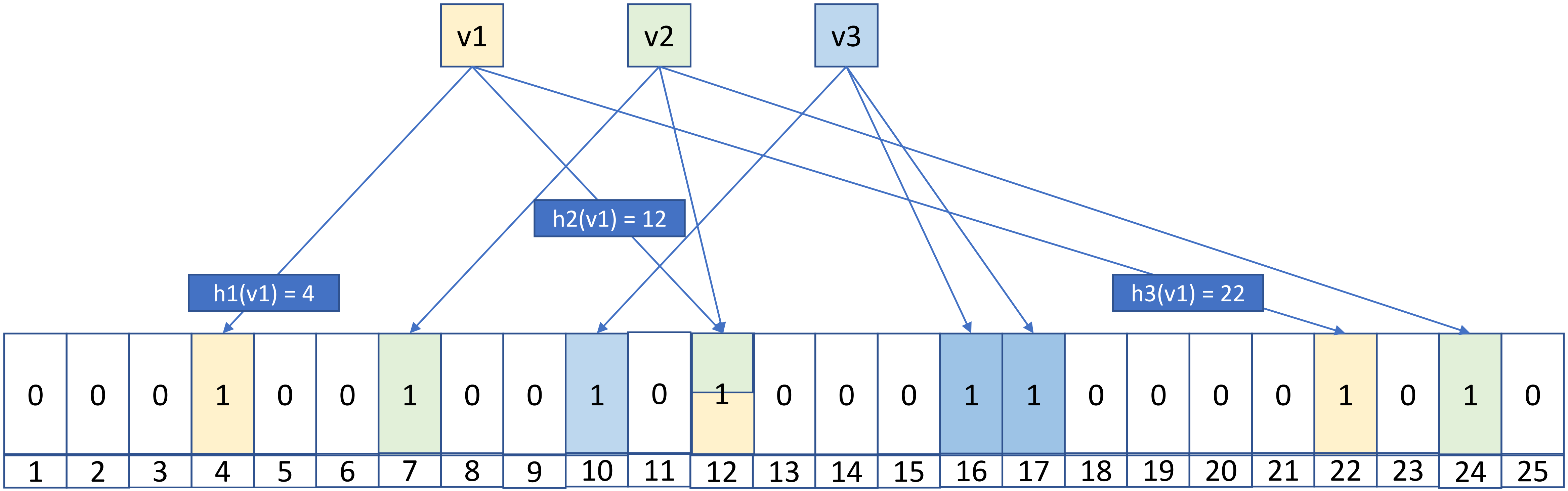
- Small probability of **false positive**. w1 ∈ S, w2 ∉ S h(w1) = h(w2)

Bloom filters

- To reduce the probability of false positives use k > 1 independent hash functions.

- How it works: S, set of n values → *const * n* bits
  calculate $h_1(v)$, $h_2(v)$ … $h_k(v)$ ∈ $[1, \mathit{const} * n]$
  set bits $h_1(v)$, $h_2(v)$ … $h_k(v)$ to 1

  Test w ∈ S → $h_1(v) = 1$ and $h_2(v) = 1$ … and $h_k(v) = 1$?



Small probability of **false positive**.

Probability of **false negative** = 0.

Bloom filters

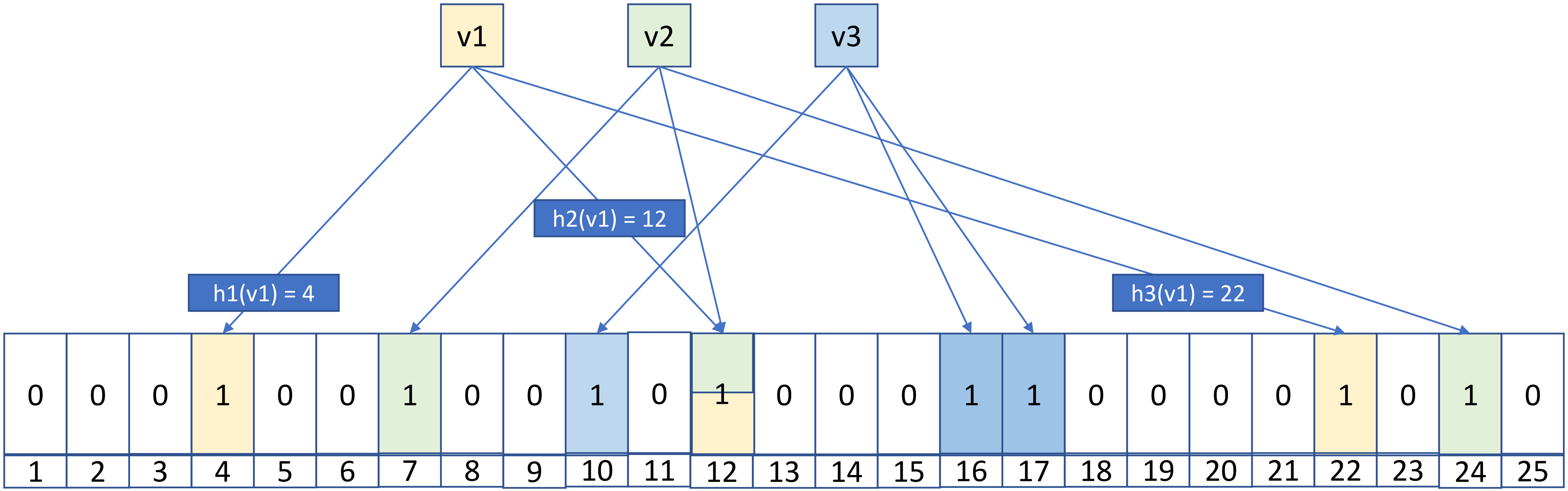
- Used only to add elements or the test membership.
- Once an element is added to the filter it cannot be removed.
- If all bits are set to 1, the probability of false positives increases.

  More space → more accuracy.
- More hash functions

  Latency → more accuracy.

Bloom filters – independent hashing

- A family of hash functions $H = \{h: U \to [1..m]\}$ is k-independent if $\forall\ (x_1, x_2 \ldots x_k) \in U^k$ and $\forall\ (y_1, y_2 \ldots y_k) \in [1..m]^k$:
    - $Pr_{h \in H}\ [h(x_1) = y_1 \wedge h(x_2) = y_2 \ldots \wedge h(x_k) = y_k] = \frac{1}{m^k}$
- $h(x_1)$ uniformly distributed.
- $h(x_1), h(x_2)\ , \ldots h(x_k)$ independent random variables.



Small probability of **false positive**.

Probability of **false negative** = 0.

false positive. Value *w: B[h1(w)] = 1 B[h2(w)] =1 … B[hk[w]] = 1*

Each hash of w equals a hash of an element in the set

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

$$P = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \quad \text{or}$$

$$P = \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

- m = 10 * n and k = 7 $\simeq$ 0,01

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

h(w) != h1(v1)

$$P = \left(1 - \boxed{\left(1 - \frac{1}{m}\right)}^{kn}\right)^{k} \quad \text{or}$$

- m = 10 * n and k = 7 ≃ 0,01

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

$$P = \left(1 - \boxed{\left(1 - \frac{1}{m}\right)^{kn}}\right)^{k} \quad \text{or}$$

h1(w) != h1(v1)
h1(w) != h1(v1)
.....
h1(w) != hn(v1)
h1(w) != h1(v2)
…
h1(w) != hn(v2)
…

- m = 10 * n and k = 7 ≃ 0,01

Bloom filters – accuracy

- m size of array, n number of elements in S, k number of hash functions.
- Probability of false positive:

$$P = \left(\boxed{1 - \left(1 - \frac{1}{m}\right)^{kn}}\right)^{k} \quad \text{or}$$

h1(w) = h1(v1)
or
h1(w) = h1(v1)
.....
h1(w) = hn(v1)
or
h1(w) = h1(v2)
...
h1(w) = hn(v2)
...

- m = 10 * n and k = 7 ≃ 0,01

Log Structured Merge-tree

Log Structured Merge Trees

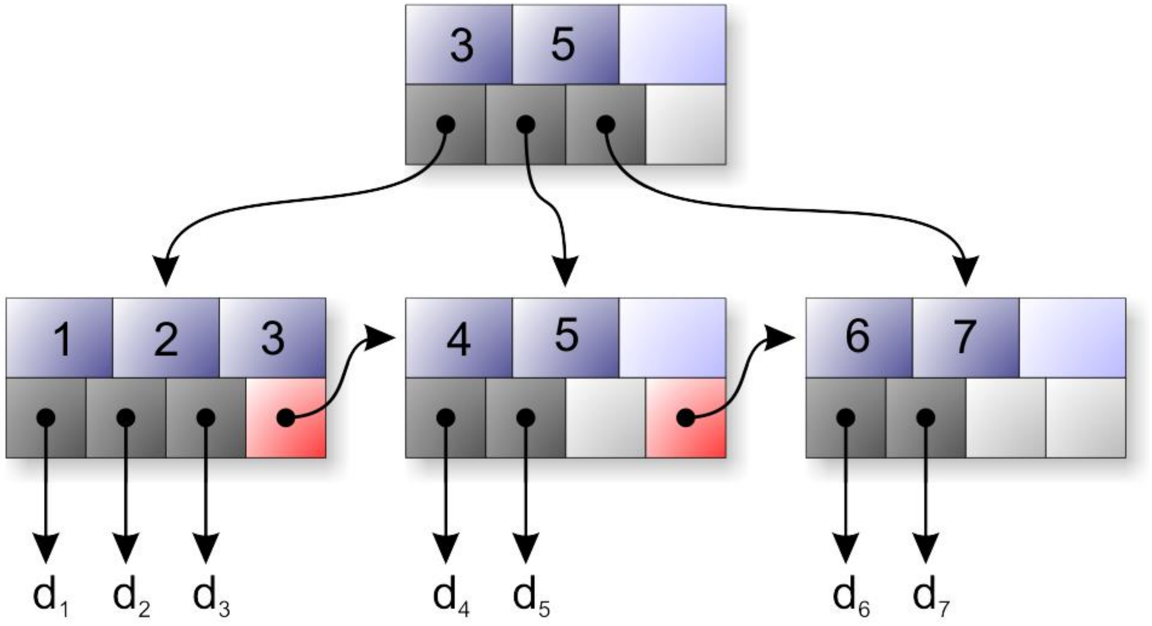
- Optimize I/O operations.

- Used by: Bigtable, LevelDB, Apache Cassandra etc.

- Data organized in B+ trees.

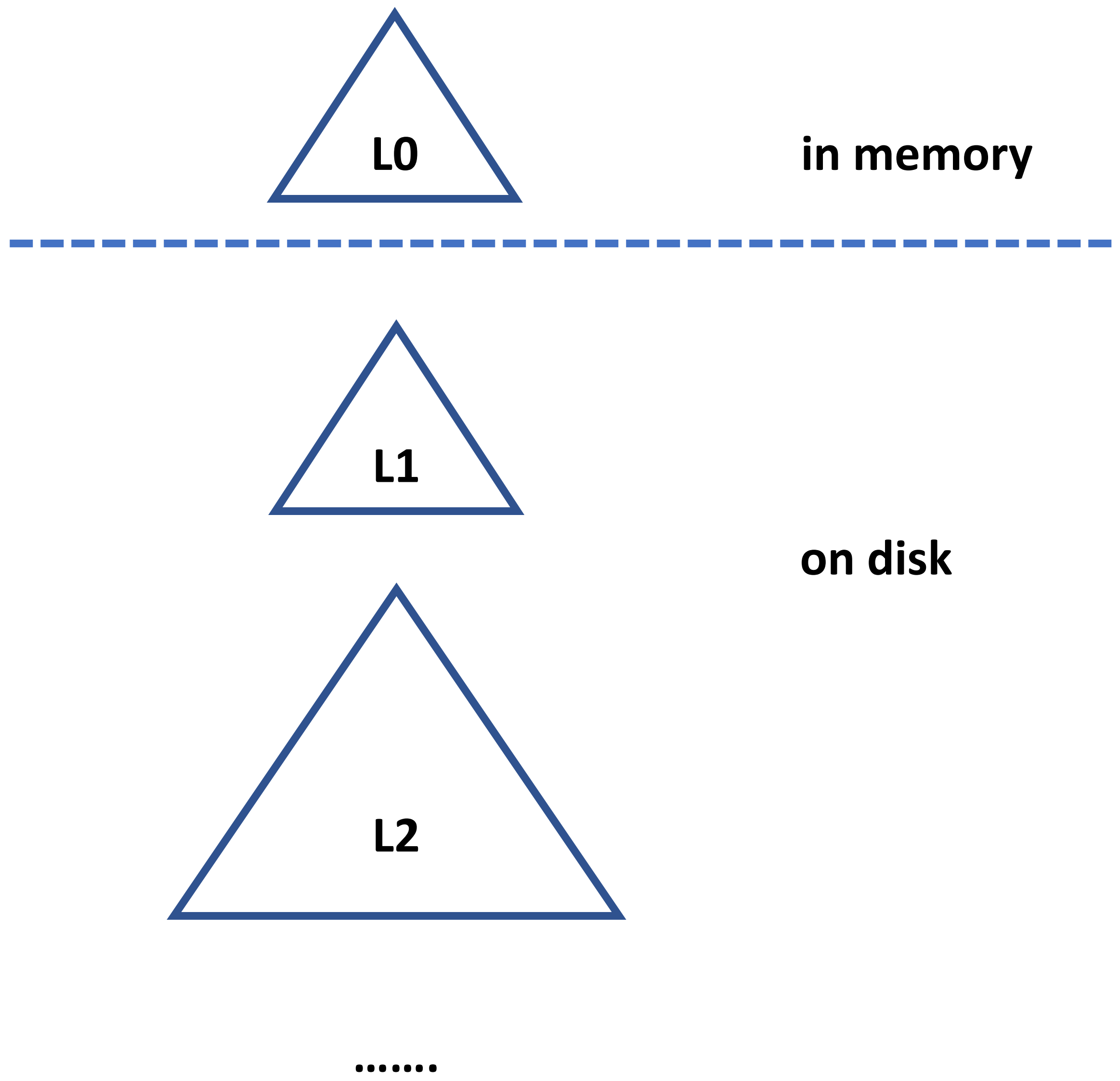
- Advantages: leaves sequentially located,
  leaves are full.

B+ tree

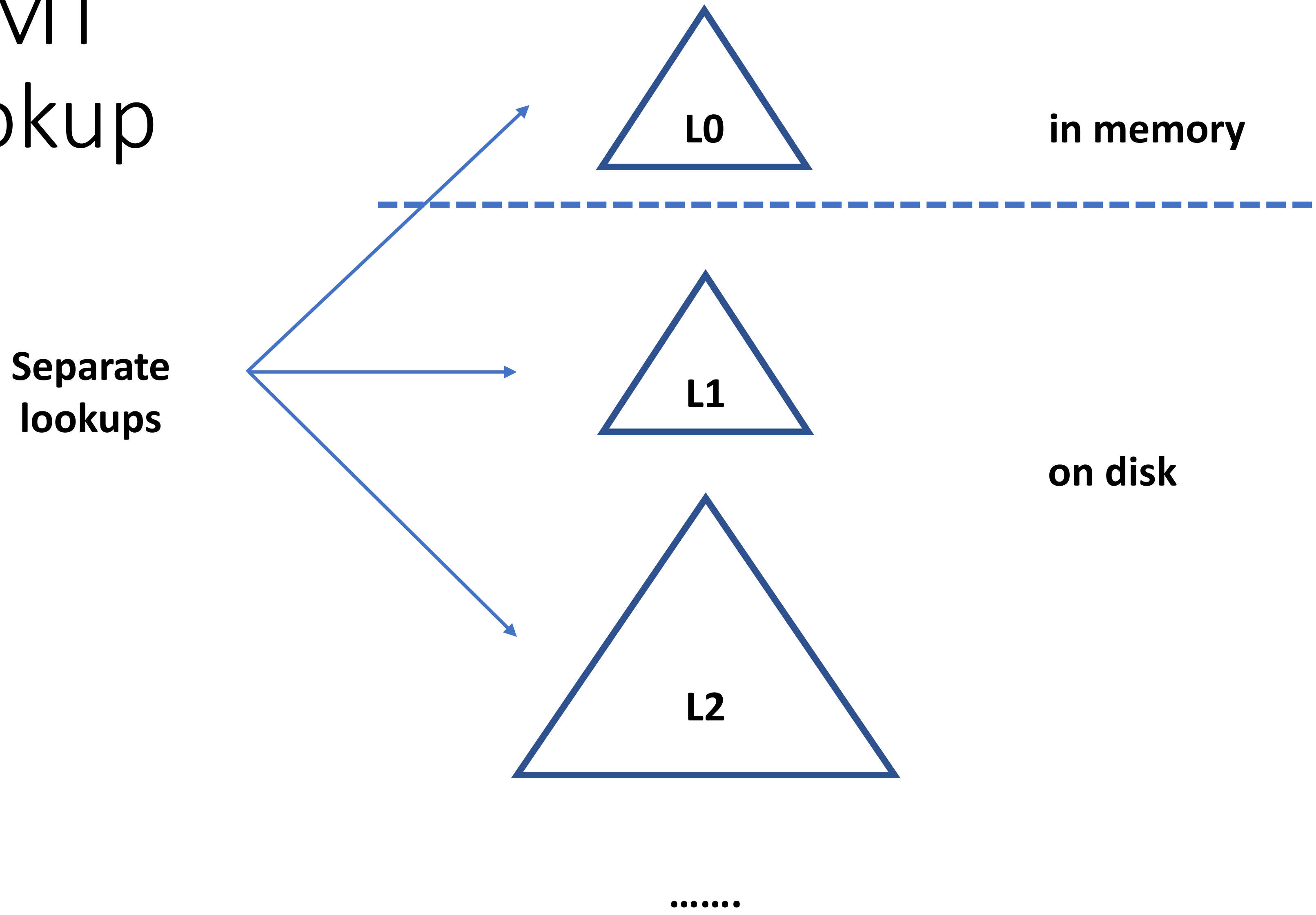


https://commons.wikimedia.org/wiki/File:Btree.png

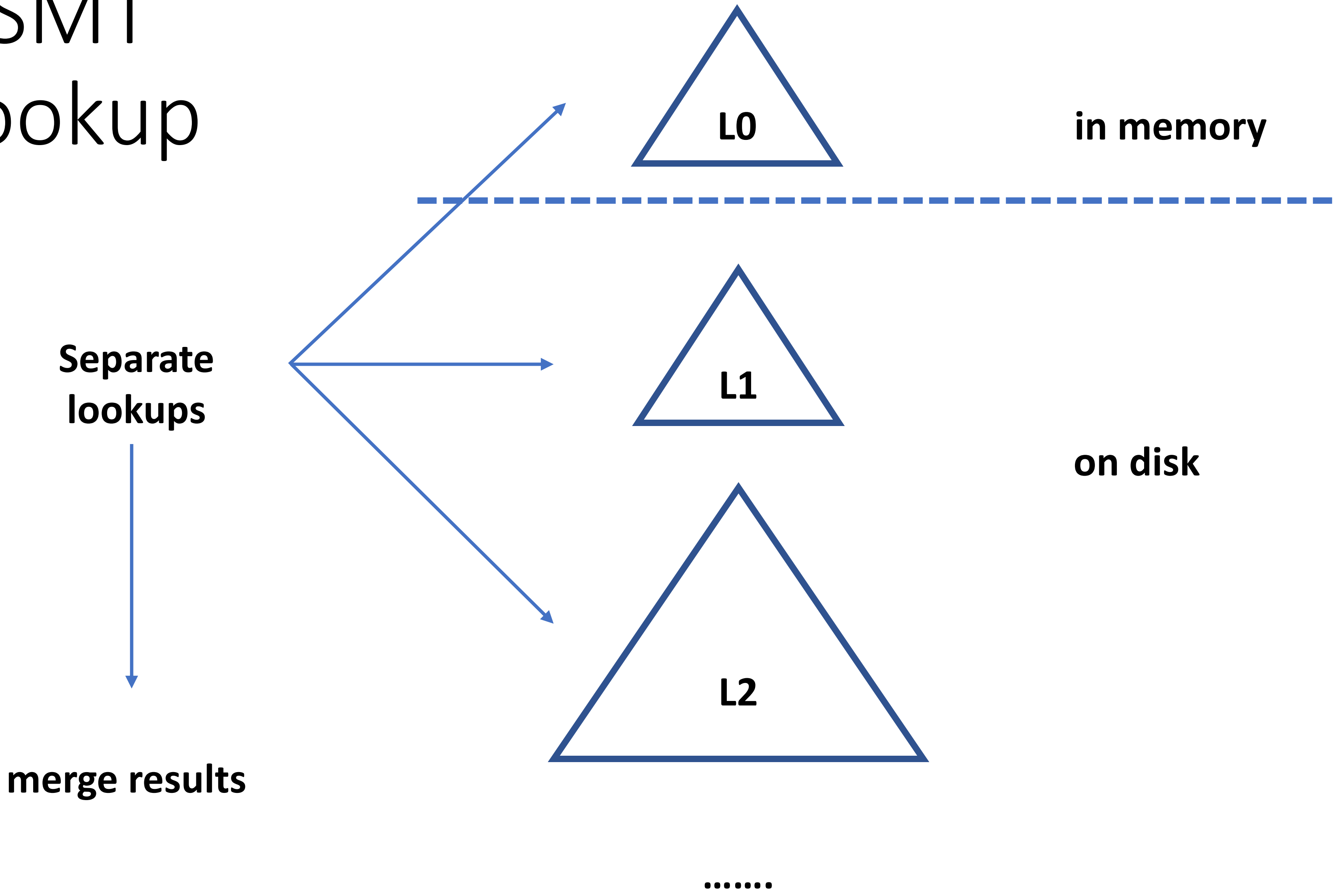
LSMT



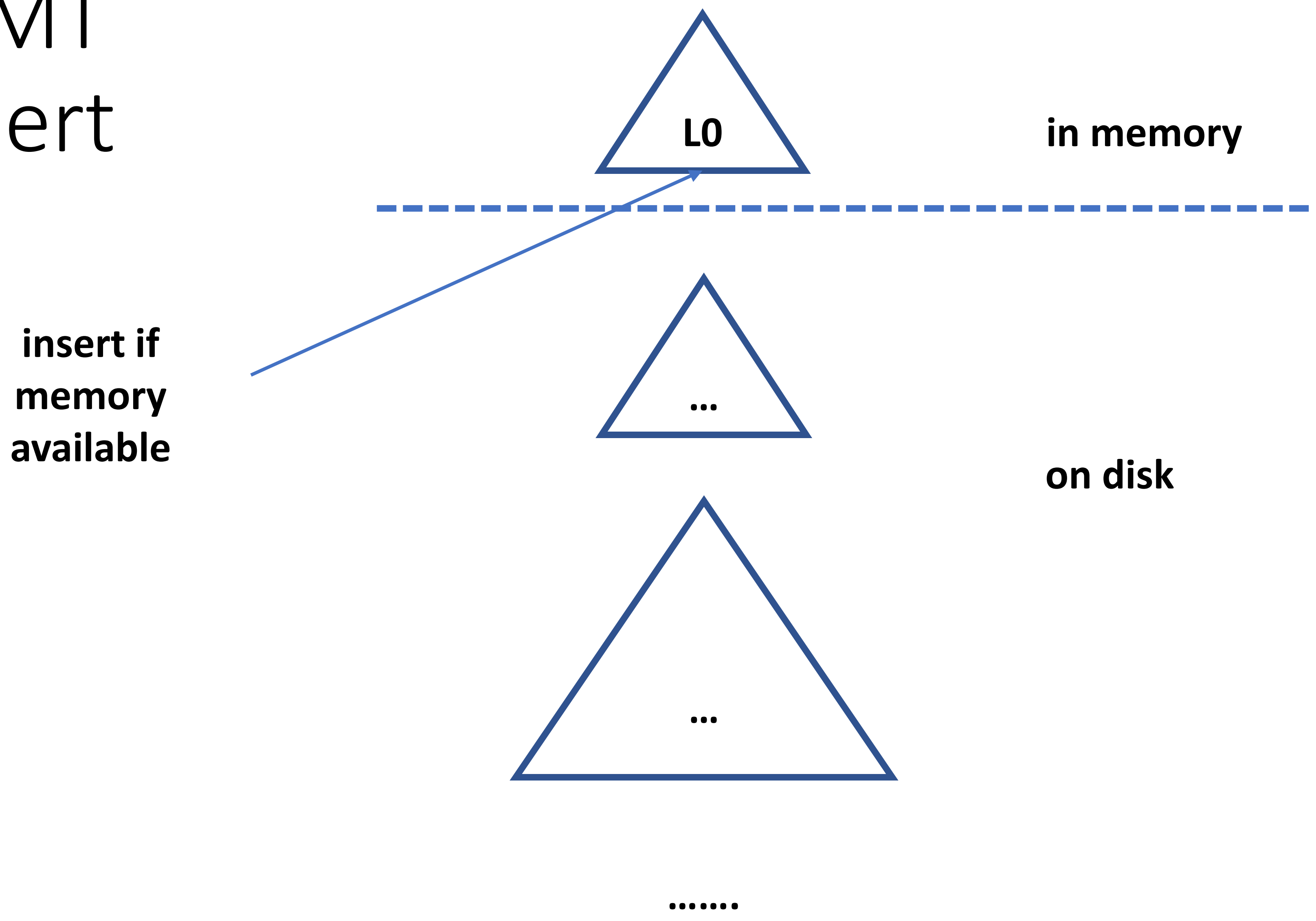
LSMT lookup



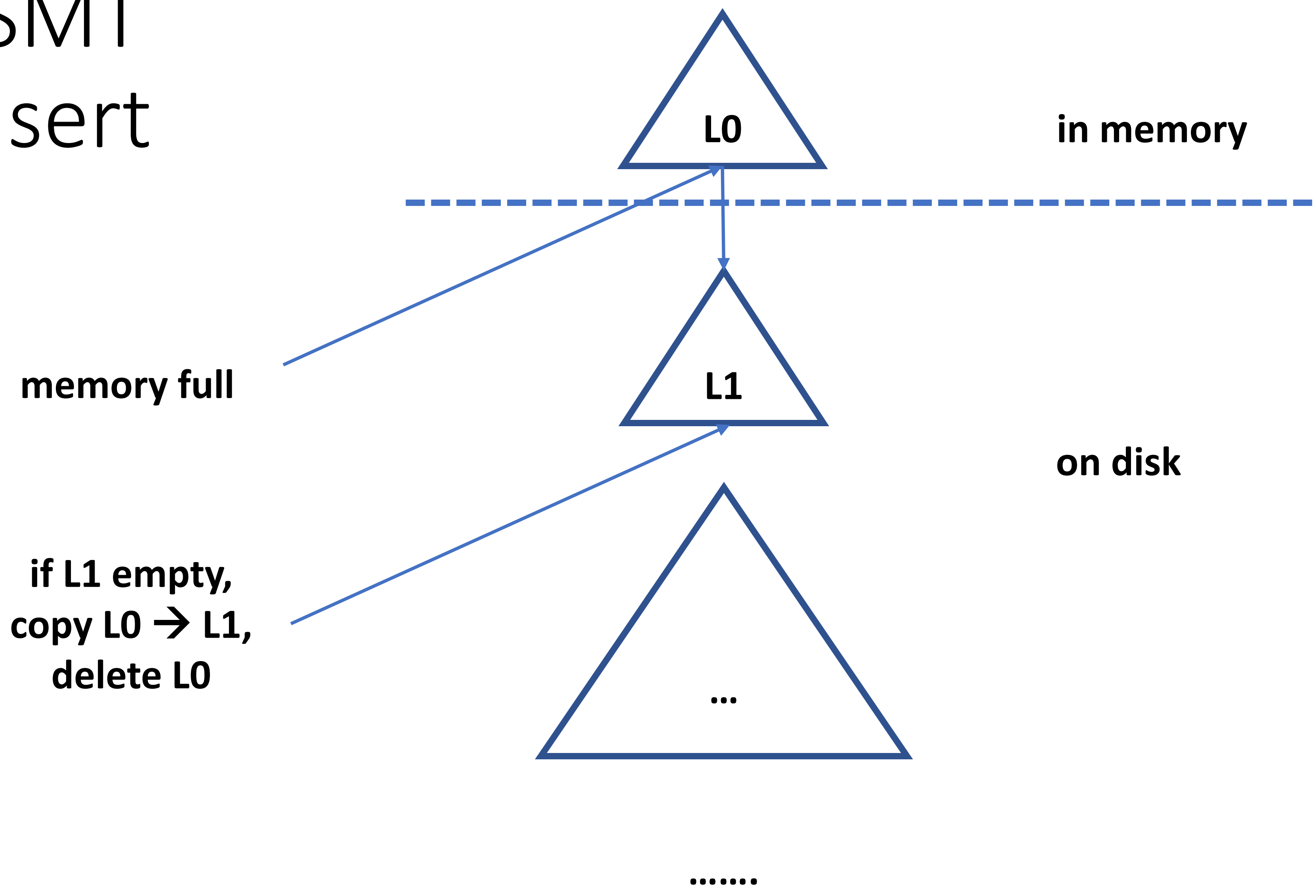
LSMT lookup



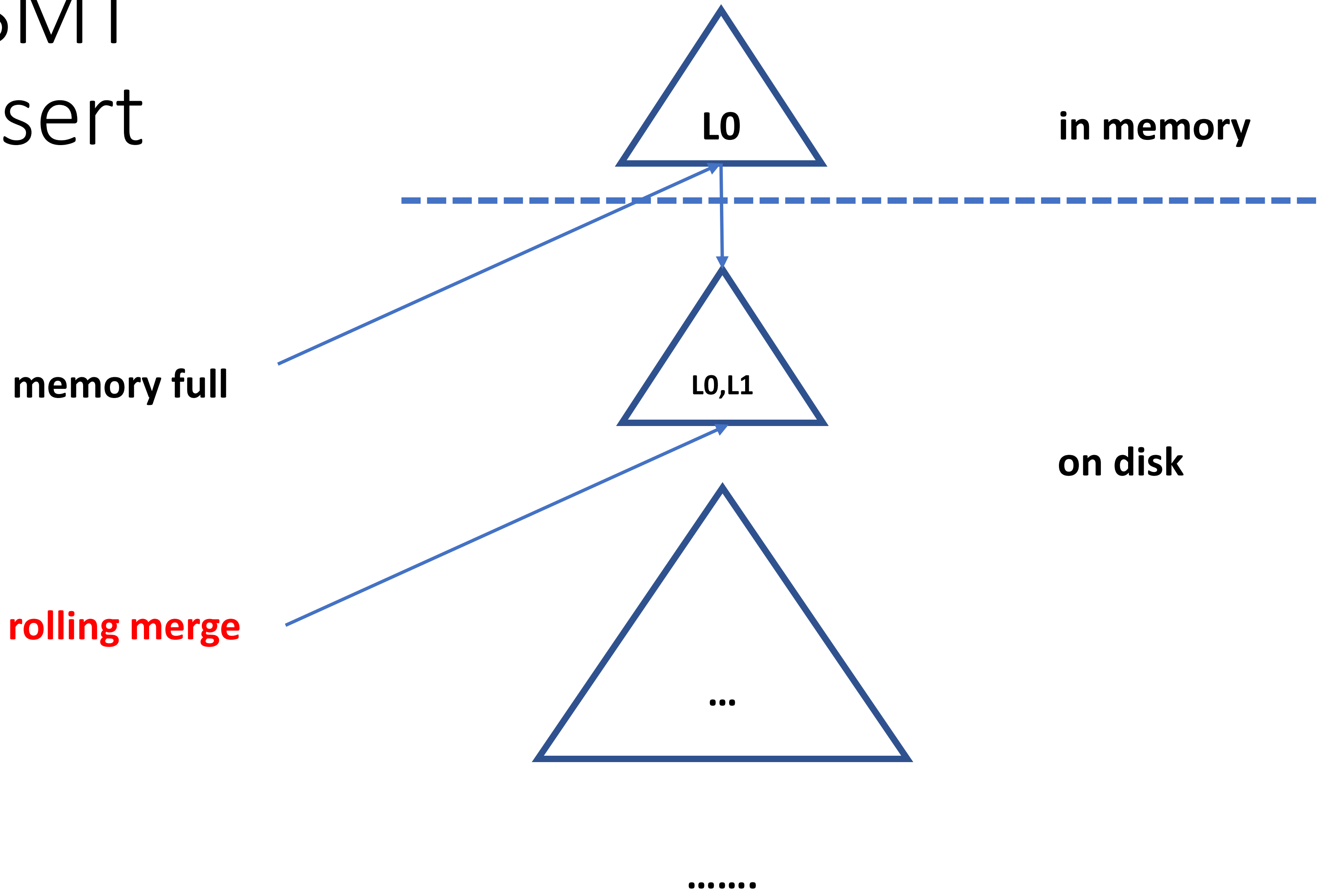
LSMT insert



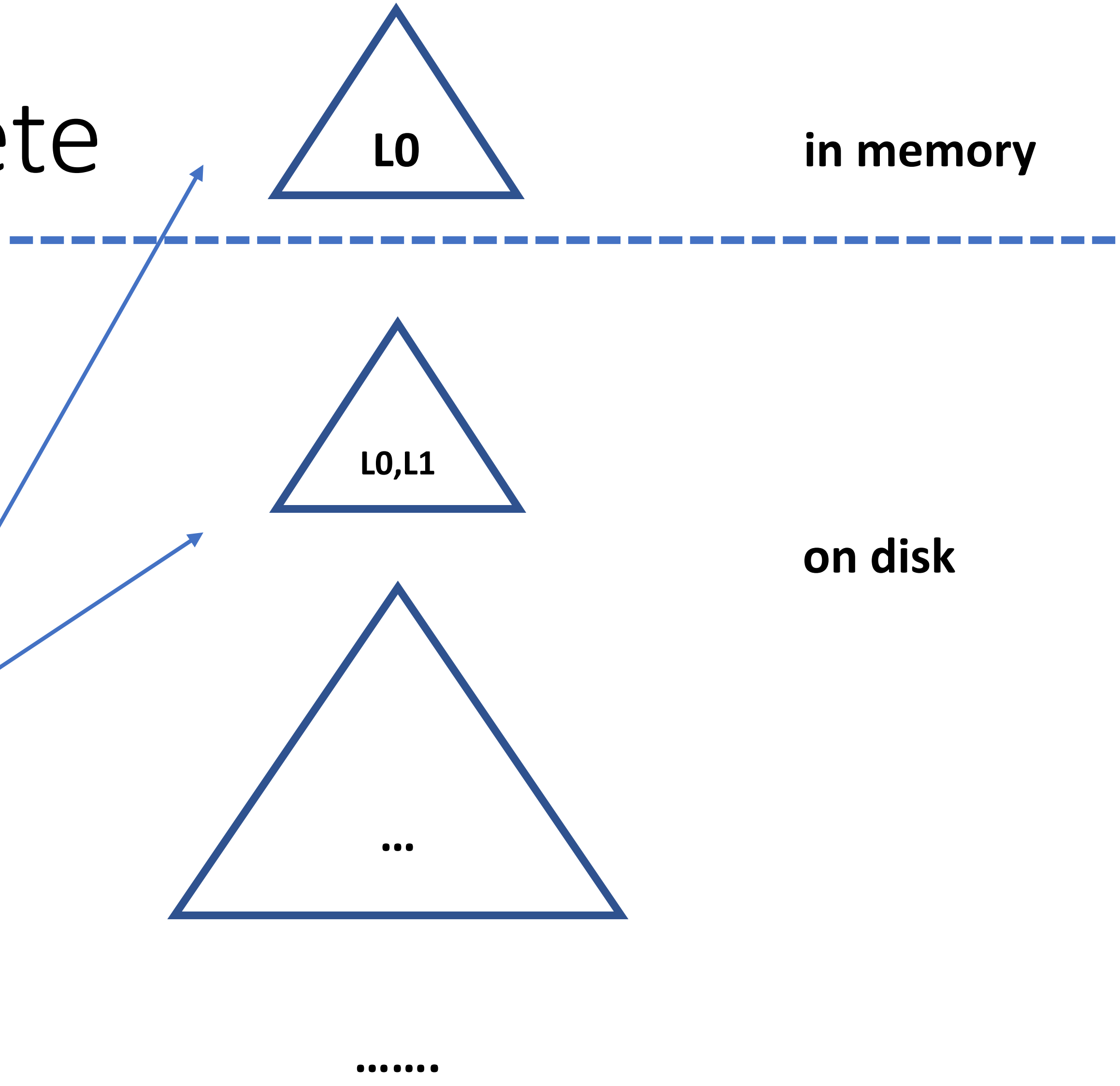
LSMT insert


L0
in memory
memory full
L1
on disk
if L1 empty,
copy L0 → L1,
delete L0
...
....

LSMT insert



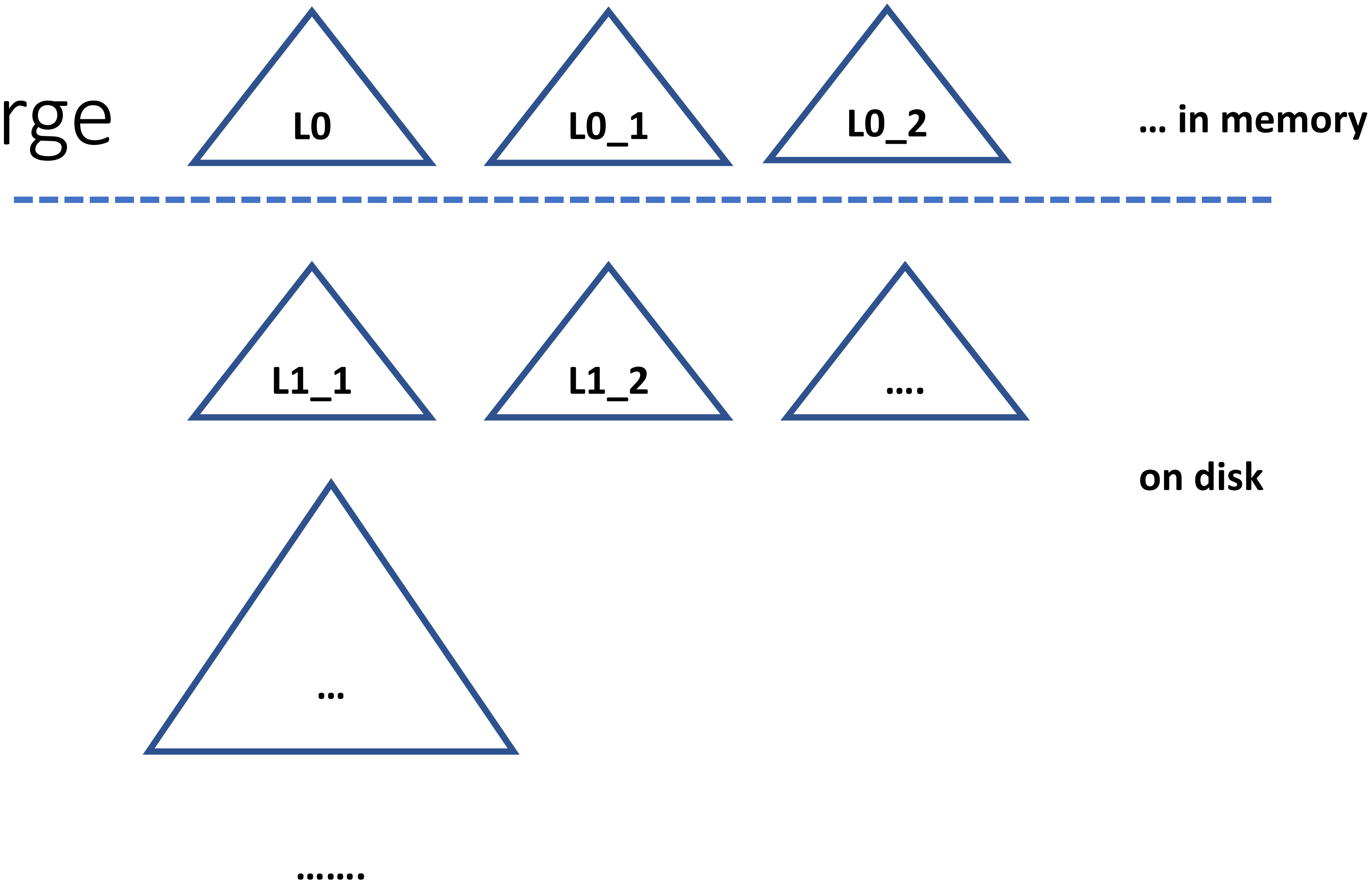
LSMT
update, delete



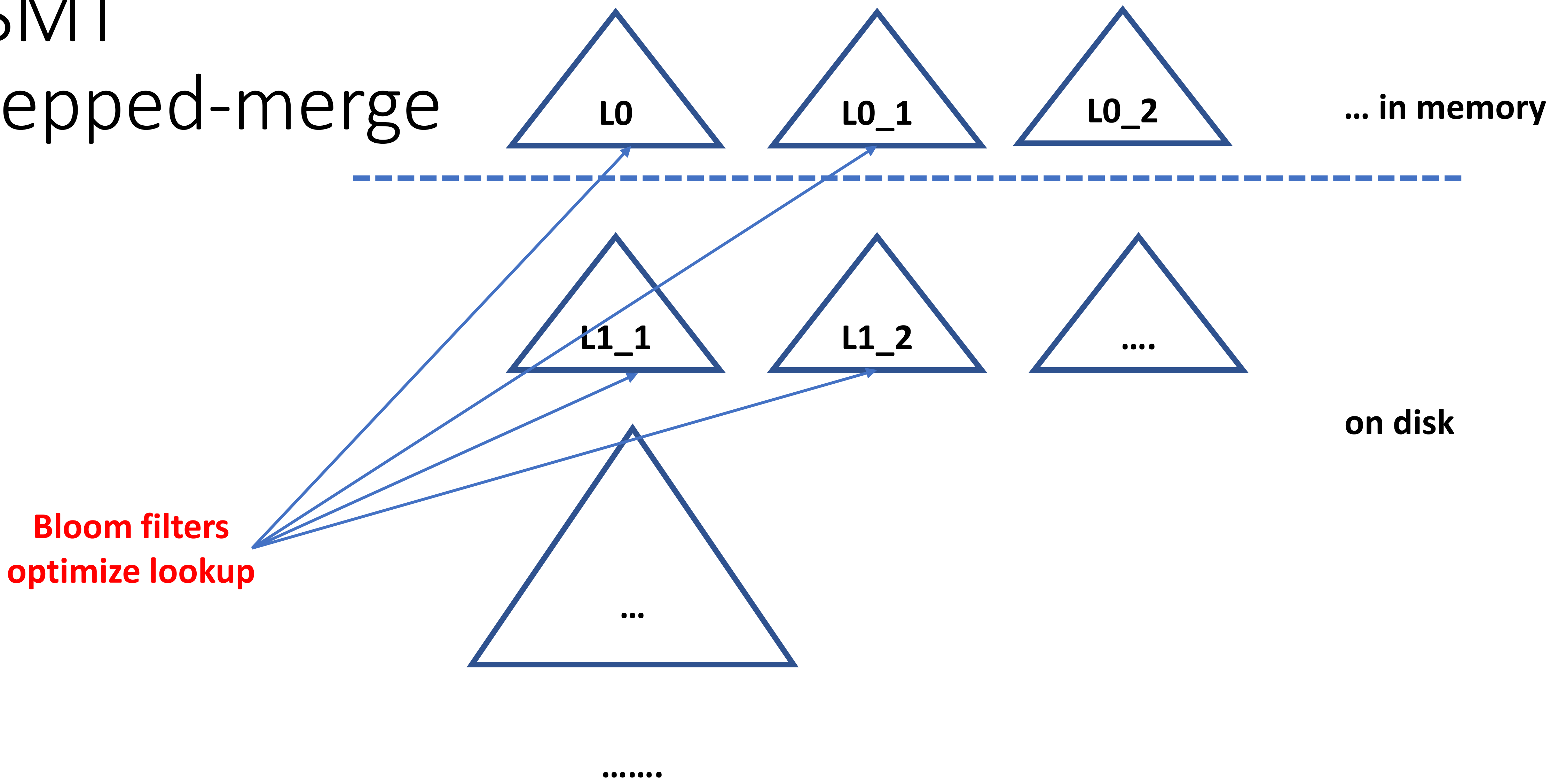
“deleted” or “updated” recoreds,

udapted merge: check if record is deleted or updated

LSMT stepped-merge



LSMT stepped-merge



Materialized views

Materialized views

- redundant data, contents can be inferred from the definition
- immediate view refresh
- deferred view refresh
- incremental update: modify only the affected parts of the materialized view

Materialized views

- Join operation
- Selection
- Projection
- Aggregation