Hash table and Inverted Indexes

Indici basati su hashing

Gli indici basati su hashing utilizzano una funzione hash per mappare chiavi di ricerca e posizioni specifiche nei bucket di memoria, ottimizzando le operazioni di ricerca e aggiornamento:

- *vantaggio*: perfetti per ricerche di uguaglianza
- *svantaggio*: non supportano in modo efficiente le ricerche su intervalli
  Si suddividono in due categorie principali:

1. **Static Hashing**: la dimensione del bucket è fissa, il che lo rende inadatto a dati dinamici;
2. **Dynamic Hashing**: tecniche come l'hasing estendibile e lineare permettono di gestire l'aumento dei dati senza degradare le prestazioni.

Hashing Statico

Definizione e funzione hash

- utilizza *N* bucket, ognuno identificato da un numero compreso tra *0* e *N-1*;
- una funzione hash *H(K)* mappa ogni chiave di ricerca *K* a un bucket specifico. Ad esempio, *H(K)* = *i*, con *i* compreso tra *0* e *N-1*;
- la funzione hash può basarsi su un prefisso o un suffisso dei bit del valore della chiave. Ad esempio, la funzione *H*, restituisce i bit più significativi (o meno significativi) del valore binario della chiave.

Inserimento

- un record con chiave *K* viene inserito nel bucket *H(K)*;
- se un bucket è pieno, viene utilizzata una **catena di overflow**, che collega più blocchi per accogliere i record aggiuntivi.

Ricerca

- per trovare un record con chiave *K*, la funzione hash calcola il bucket *H(K)*;
- il sistema cerca nei blocchi del bucket (inclusi quelli di overflow, se presenti).

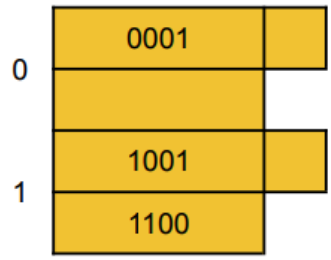
Cancellazione

- quando un record viene eliminato, si possono rimuovere blocchi di overflow, ma questo richiede attenzione per garantire che i dati rimanenti siano correttamente gestiti

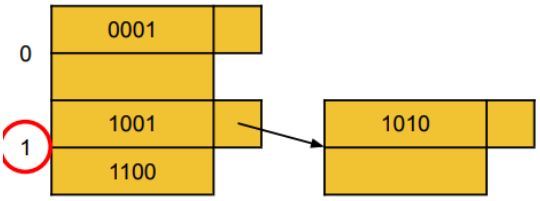
Esempio:
Supponiamo di avere *N* = *2* bucket, *i* = *1* bit per la funzione hash H_1. I bucket sono numerati *0* e *1*, e ogni bucket contiene un blocco.
Esempi di chiavi e bucket:

- chiave *K* = 0001 → $H_1(0001) = 0$ (va nel bucket 0);

- chiave K = 1100 → $H_1(1100) = 1$ (va nel bucket 1);
- chiave K = 1001 → $H_1(1001) = 1$ (va nel bucket 1);

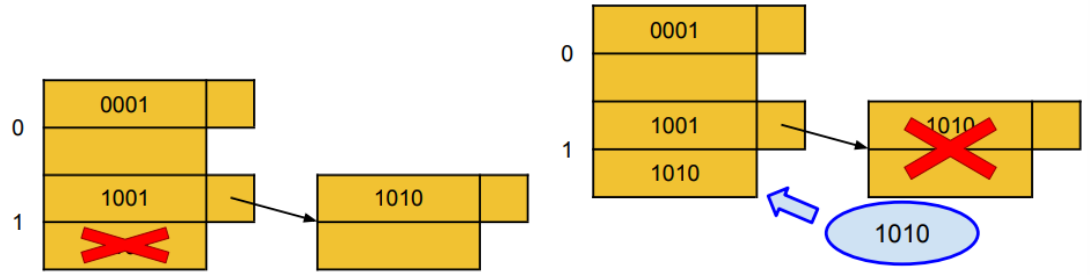


Supponiamo di voler aggiungere una nuova chiave K che genera però un overflow nel bucket, ad esempio K = 1010, aggiungiamo allora un blocco di overflow.



Supponiamo invece di voler rimuovere la chiave K = 1100, questo ci consente:

- in presenza di blocchi di overflow, di eliminarli per liberare spazio;
- di rimuovere un blocco se dopo l'eliminazione questo rimane vuoto.



Efficienza dell'hashing statico:

Dipende da:

1. **Numero di bucket**:
   - se i bucket sono sufficienti a contenere i dati senza generare overflow, la ricerca richiede un solo accesso al disco;
   - catene di overflow lunghe degradano le prestazioni, richiedendo accessi multipli al disco.
2. **Distribuzione delle chiavi**:
   - una funzione hash ben progettata distribuisce uniformemente le chiavi nei bucket;
   - una distribuzione sbilanciata (ad esempio, molte chiavi che generano lo stesso valore hash) causa lunghe catene di overflow.

Limiti dell'hashing statico

1. **Dimensione fissa**: non è possibile modificare il numero di bucket, questo rende il metodo inadatto per dataset in crescita o che cambiano frequentemente.
2. **Gestione inefficiente dello spazio**: se la dimensione dei dati è molto inferiore al numero di bucket, alcuni rimangono vuoti, sprecando spazio.
3. **Mancanza di supporto per ricerche su intervalli**: l'hashing statico è progettato per ricerche di uguaglianza (es. "trova il record con chiave K), ma non supporta efficientemente ricerche su

intervalli (es. "trova tutti i record con chiavi tra *K1* e *K2*").

Hashing Estendibile

E' una tecnica di hashing dinamico progettata per superare i limiti dell'hashing statico, come la dimensione fissa dei bucket e la gestione inefficiente di dataset dinamici.

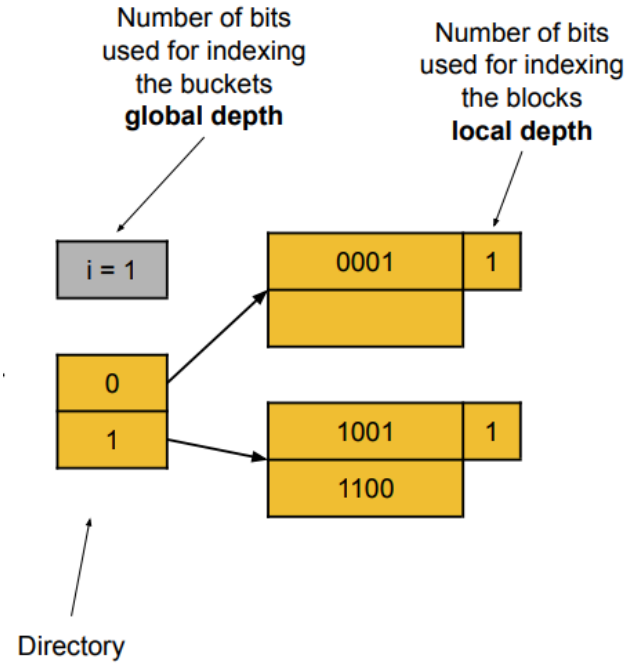
Principi base

L'hashing estendibile utilizza una **directory di puntatori** ai bucket che può crescere dinamicamente. Ogni bucket è identificato da un **prefisso di bit** delle chiavi memorizzare:

- **profondità globale** (GD): numero di bit usati per indirizzare i bucket nella directory;
- **profondità locale** (LD): numero di bit usati per distinguere i record all'interno di un bucket.

Struttura

1. Directory:
   - contiene 2^{GD} puntatori ai bucket;
   - la dimensione della directory aumenta raddoppiando quando GD viene incrementato.
2. Bucket:
   - ogni bucket è associato a un prefisso di lunghezza LD;
   - un bucket può essere condiviso da più voci della directory se LD < GD.



Operazioni:

Inserimento:

- la funzione hash calcola un indice basato su GD bit della chiave;
- se il bucket corrispondente ha spazio, il record viene inserito;
- se il bucket è pieno:
  1. Si controlla se LD < GD:
     - il bucket viene **diviso**. I record vengono ridistribuiti tra due nuovi bucket in base al bit successivo della chiave;

- LD del bucket viene incrementato.

2. Se LD = GD:
   - la directory viene raddoppiata;
   - GD viene incrementato, e i puntatori nella directory vengono aggiornati per riflettere la nuova configurazione.

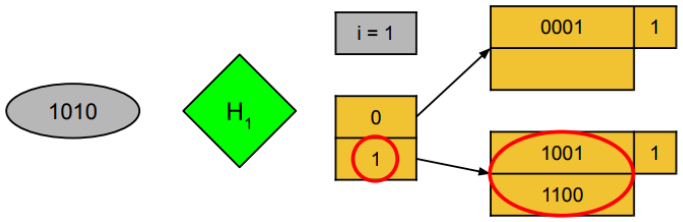
Ricerca:

- la funzione hash calcola l'indice del bucket nella directory utilizzando GD bit;
- si accede direttamente al bucket corretto tramite il puntatore della directory.

Cancellazione:

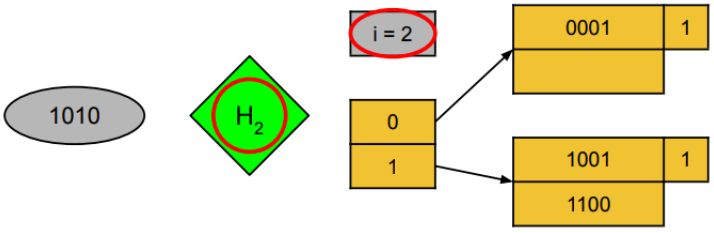
- il record viene eliminato dal bucket corrispondente;
- se un bucket diventa vuoto, può essere combinato con un altro bucket (se condividono lo stesso prefisso e LD è maggiore del minimo necessario).

Esempio:
Chiamiamo GD → i e LD → j.
Supponiamo di avere una directory iniziale con GD = 1 (1 bit) e due bucket (0 e 1). Ogni bucket può contenere al massimo due chiavi. Ora vogliamo inserire la chiave K_2 = 1010:

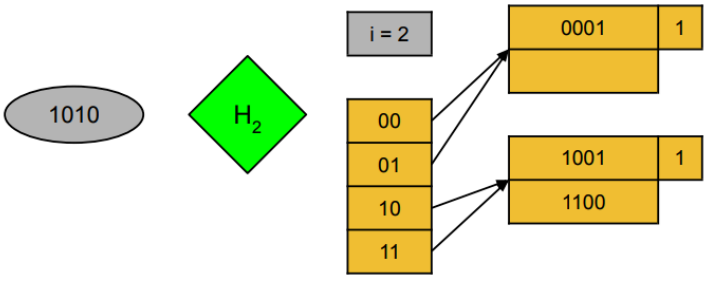


There is no room!

Controlliamo i valori i e j, dal momento che i = j dobbiamo incrementare i:

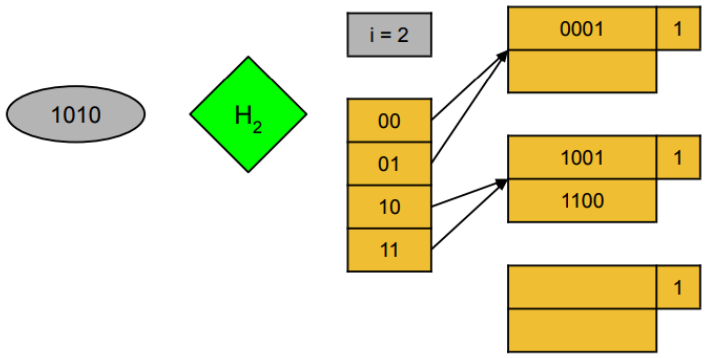


Adesso i = 2 e la directory raddoppia; il bucket 1 viene diviso in due nuovi bucket (10 e 11) :

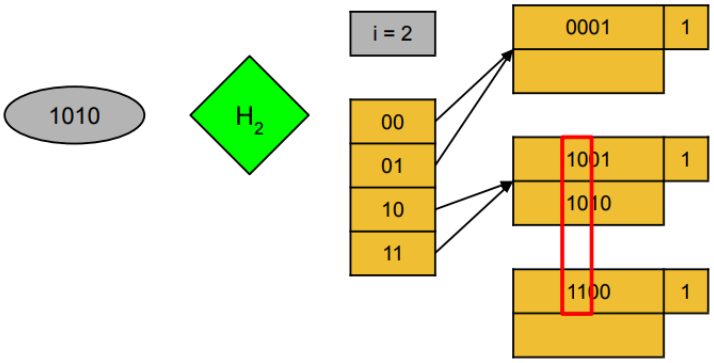


j < i

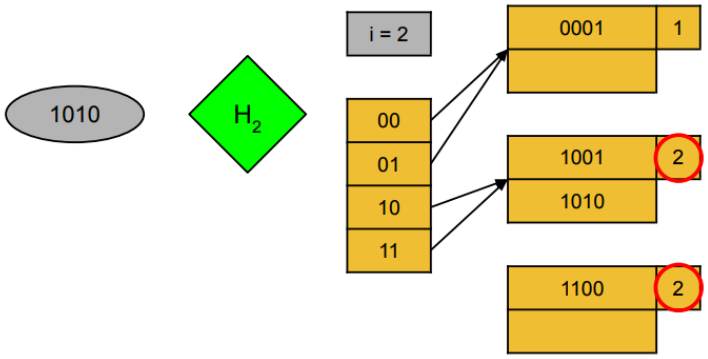
il blocco B viene ulteriormente diviso:



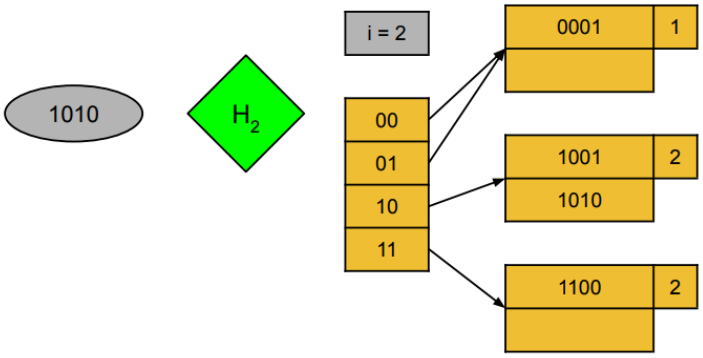
i record vengono ridistribuiti utilizzando j+1 bit:



il valore di j viene incrementato di 1:



infine, la directory viene aggiornata con il puntatore al nuovo blocco:



Gestione delle profondità:

1. **Profondità globale (GD)**:
   - indica quanti bit vengono usati dalla directory per reindirizzare i bucket;
   - se un bucket pieno ha LD = GD, la directory deve essere raddoppiata.
2. **Profondità locale (LD)**:
   - determina quanti bit del prefisso sono condivisi dalle chiavi in un bucket;
   - se un bucket viene diviso, LD viene incrementato.

Vantaggi e Svantaggi:

Vantaggi:

- **espansione dinamica**: la directory cresce solo quando necessario;
- **efficienza**: la ricerca rimane veloce anche con dataset in crescita;
- **riduzione degli overflow**: i bucket pieni vengono divisi anziché usare blocchi di overflow.
  Svantaggi:
- la dimensione della directory può diventare molto grande se i dati sono distribuiti in modo non uniforme;
- aumento della complessità rispetto all'hashing statico.

Hashing lineare

E' una tecnica di hashing dinamico progettata per gestire dataset in crescita in modo flessibile, evitando gli inconvenienti di dimensioni fisse e riducendo la necessità di raddoppiare bruscamente la memoria, come nell'hashing estendibile.

Concetti fondamentali:

Struttura base:

- inizia con un numero *n* di bucket, organizzati in ordine sequenziale;
- i dati vengono distribuiti tra i bucket utilizzando una funzione hash h(k);
- a differenza dell'hashing statico, la struttura può crescere gradualmente.
  Hashing e split:

1. **Funzione hash**:
   - inizialmente, si usa una funzione hash h_0(k) = k mod 2^i, dove *i* è il livello corrente di suddivisione;
   - man mano che i bucket si riempiono, alcuni vengono divisi e si utilizza una funzione hash h_1(k) = k mod 2^{i+1} per ridistribuire i dati.
2. **Gestione dei bucket**:
   - l'hashing lineare mantiene un puntatore *P* che identifica quale bucket deve essere diviso successivamente;
   - quando un bucket è pieno, il puntatore *P* avanza e il bucket viene suddiviso.

Operazioni:

Inserimento

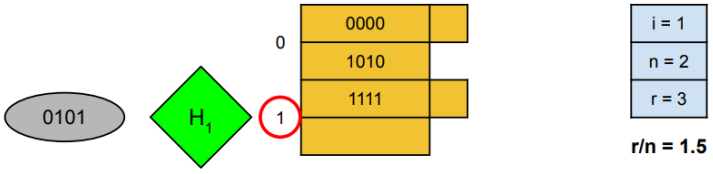
1. *conta dei record e dei bucket*:
   - si contano il numero di record *r* e il numero di bucket *n* attualmente utilizzati nella tabella hash.
2. *controllo della condizione di split*:
   - se il rapporto r/n (numero medio di record per bucket) supera la soglia di 1.7, si aggiunge un nuovo bucket, $(n+1)$-esimo bucket.
3. *divisione dei bucket*:
   - su utilizza una funzione hash H_i, che serve a distribuire i record ne bucket;

- tutti i bucket fino a 2^{i-1}-esimo vengono suddivisi secondo l'ordine in cui sono stati creati, indipendentemente da quale bucket abbia causato la divisione.

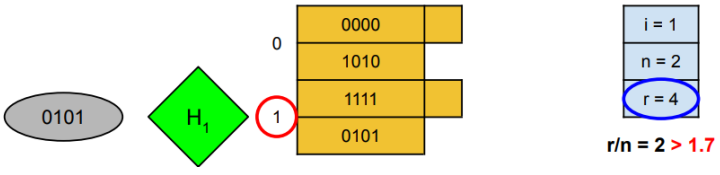
4. *cambio della funzione hash*:
   - se *n* supera 2^i, ovvero il numero massimo gestibile dall'attuale funzione hash H_i, allora si passa alla funzione hash successiva H_{i+1}, e il processo di split ricomincia dal primo bucket.
5. **inserimento normale**:
   - se la funzione hash $H_i(K)$ restituisce *m*, dove $m < n$:
     - il record con chiave K viene inserito nel bucket *m*;
     - se il bucket *m* è pieno, si crea un overflow block (struttura temporanea per gestire l'eccesso).
6. **inserimento durante lo split**:
   - se $H_i(K)$ restituisce *m*, ma $m \leq n$ (il bucket *m* è stato "spostato" a causa dello split):
     - si calcola un nuovo bucket $m' = (m - 2^{i-1})$ per posizionare il record;
     - anche qui, se il bucket è pieno, si crea un overflow block.
7. **incremento e gestione dello split**:
   - se il rapporto $r/n > 1.7$, si procede con lo split:
     - si controlla se $n = 2^i$. In tal caso, si aumenta l'indice *i* della funzione hash.
   - si esegue lo split:
     1. si calcola il nuovo bucket $n_2 = a_1a_2 \ldots a_i$, dove $a_1 = 1$;
     2. si trasferiscono i record dal bucket *m* al nuovo bucket *n*, seguendo un controllo sui bit della funzione hash;
     3. si aggiunge il nuovo bucket *n*.
   - si incrementa *n* di 1.

esempio:
vogliamo inserire la chiave 0101, *N* è il numero di bucket ($2^{i-1} < N \leq 2^i$):

- se $h_1(K) = m < n$, la chiave di ricerca viene inserita nel bucket m;
- se $h_1(K) = m \geq n$, la chiave di ricerca viene inserita nel bucket $m - 2^{i-1}$.
  $m = h_2(0101) = 1_2 = 1_{10} \Rightarrow m < n$

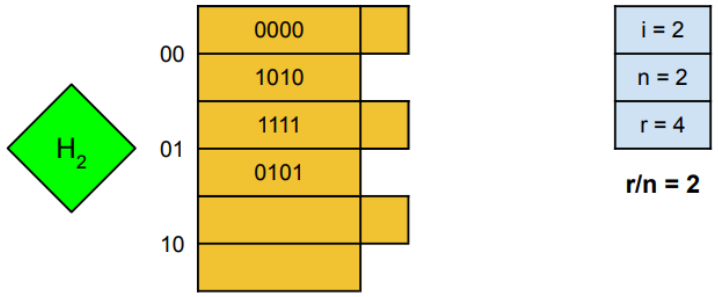


inseriamo la chiave nel bucket corretto e aggiorniamo *r*:



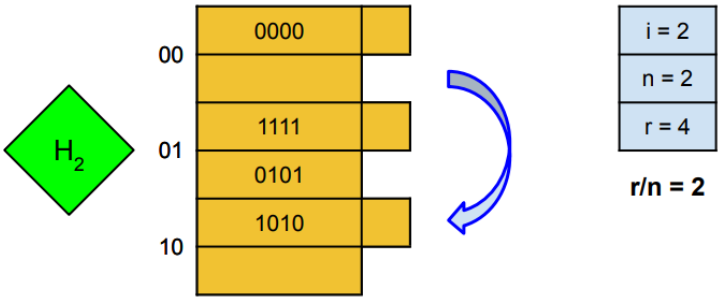
adesso però il rapporto $r/n > 1.7$ e $n = 2^i$, allora incrementiamo l'indice *i*:

- $n_2 = a_1a_2 \ldots a_i$ con $a_1 = 1 \Rightarrow n_2 = 10$
- il primo bit in *n* viene "pulito" e memorizzato in *m*
  $a_1a_2 \ldots a_i \rightarrow 0a_2 \ldots a_i \Rightarrow m_2 = 00$
  aggiungiamo il bucket $n_2 = 10$:

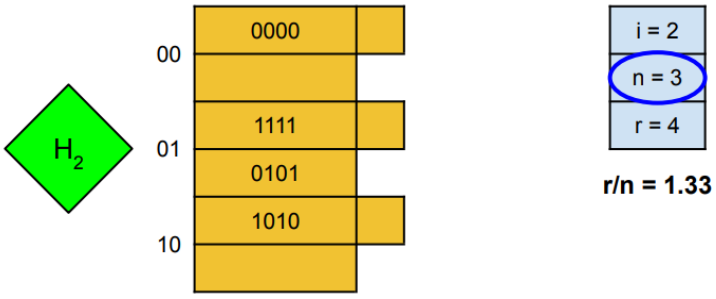


adesso spostiamo i record del bucket $m_2 = 0a_2a_3 \ldots a_i$ che hanno l'i-esimo bit più a destra uguale a 1:

- $n = 2_{10} = 10_2 (\equiv 1a_2a_3 \ldots a_i) \rightarrow 10_2$ identifica il nuovo bucket
- spostiamo i record da $00_2 (\equiv 0a_2a_3 \ldots a_i)$ a 10_2

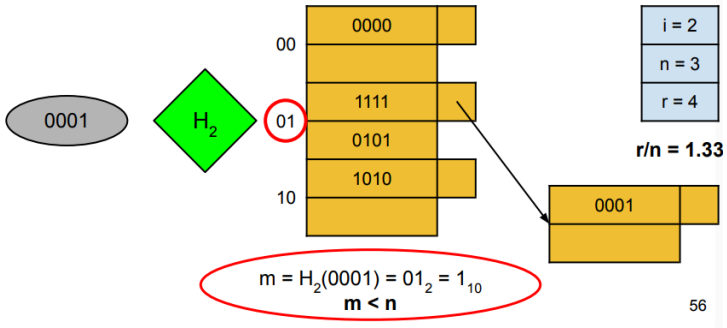


incrementiamo n:

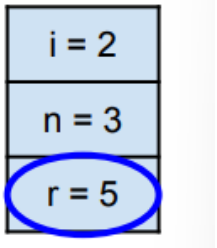


supponiamo adesso di voler inserire la chiave K = 0001

$m = H_2(0001) = 01_2 = 1_{10} \Rightarrow$ m < n



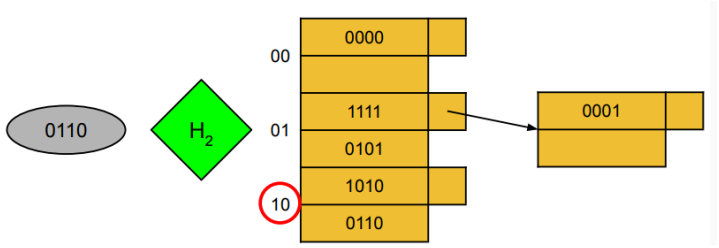
dal momento che il bucket 01 è pieno viene creato un blocco di overflow,



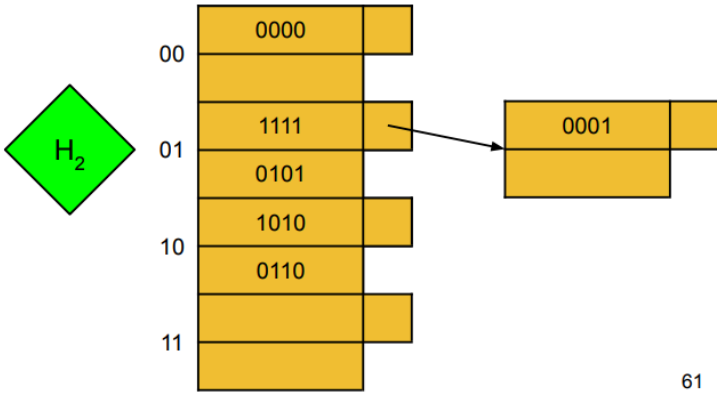
inoltre viene incrementato r: **r/n = 1.66 < 1.7**

ora supponiamo di voler aggiungere la chiave K = 0110

$m = H_2(0110) = 10_2 = 2_{10} \Rightarrow$ m < n

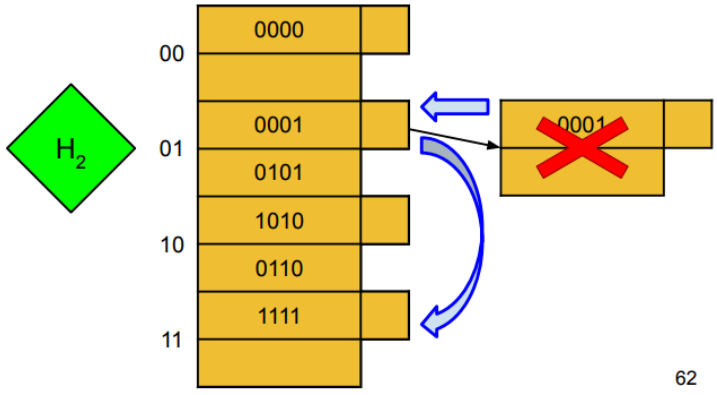


adesso però il rapporto r/n = 2 > 1.7 e $n \neq 2^i$, perciò non serve incrementare i ma basta aggiungere il bucket $n_2 = 11$:

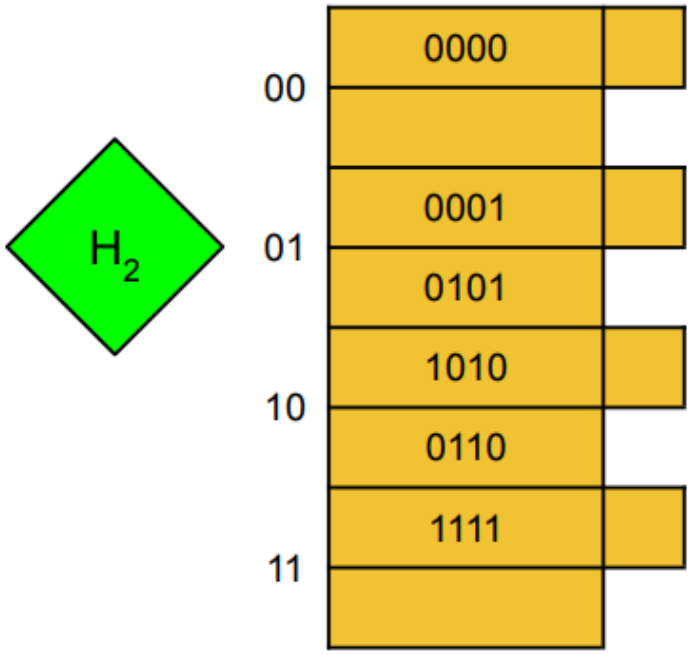


spostiamo nell'n-esimo bucket tutti i record dal bucket $0a_2a_3 \dots a_i$ che hanno l'i-esimo bit più a destra uguale a 1:

- $n = 3_{10} = 11_2 (\equiv 1a_2a_3 \dots a_i)$
- spostiamo da $01_2 (\equiv 0a_2a_3 \dots a_i)$ a 11_2.



situazione finale:

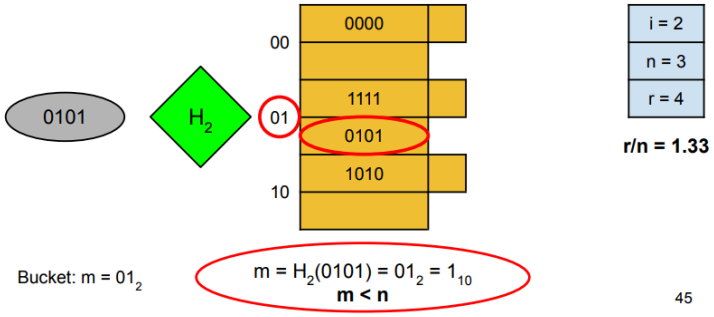


il rapporto $r/n = 1.5 < 1.7$.

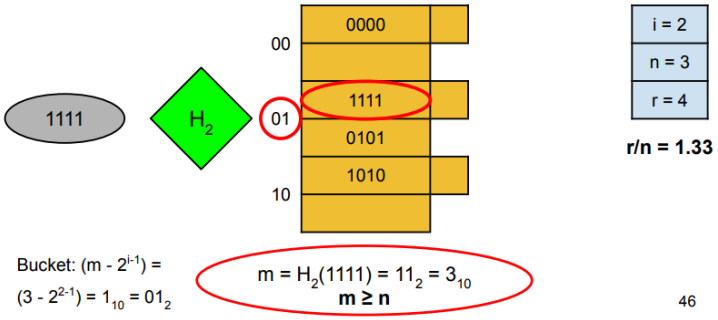
Ricerca

1. Calcola $h_0(k)$ e verifica se il record si trova nel bucket corrispondente;
2. in caso di overflow, cerca anche nel bucket creato durante la suddivisione.
   Esempio:
   cerchiamo la chiave K=0101;
   N numero di buckets (dove $2^{i-1} < N \leq 2^i$).

- se $h_i(K) = m < n$, la chiave di ricerca è nel bucket m:



- se $h_i(K) = m \geq n$, la chiave di ricerca è nel bucket $m - 2^{i-1}$:



Cancellazione

- elimina il record dal bucket;
- se un bucket diventa sottoutilizzato, non è previsto un accorpamento immediato, per mantenere l'efficienza.

Processo di suddivisione (Split)

Il processo di suddivisione dei bucket è il fulcro dell'hashing lineare.

Indici invertiti

Information Retrieval

L'information retrieval è il processo di ricerca e identificazione di documenti o informazioni rilevanti in un insieme di dati non strutturati.
applicazioni comuni:

- motori di ricerca (es. google);
- sistemi di ricerca per biblioteche digitali;
- sistemi di gestione documentale.
  La ricerca di documenti basata su parole chiave è un problema complesso perché i documenti sono *non strutturati*, a differenza dei dati strutturati in tabelle.
  Per i documenti non strutturati possiamo utilizzare tecniche come gli *indici invertiti*:
- mappano parole chiave ai documenti in cui appaiono;
- consentono di eseguire ricerche rapide su grandi raccolte di testo.

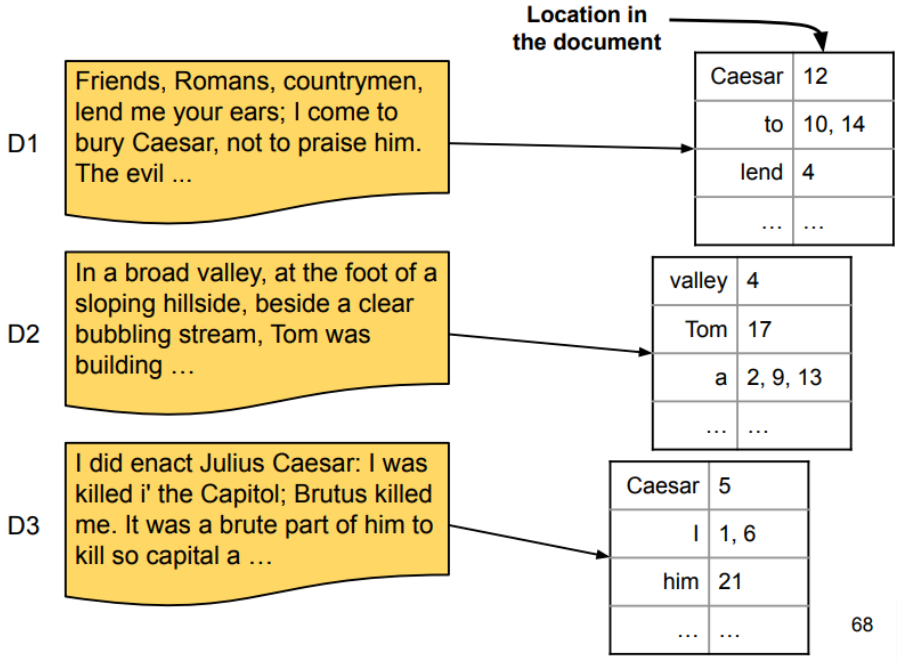
Un **indice invertito** è una struttura dati utilizzata per cercare rapidamente documenti non strutturati, come testo libero, basandosi su query che contengono parole chiave o frasi:

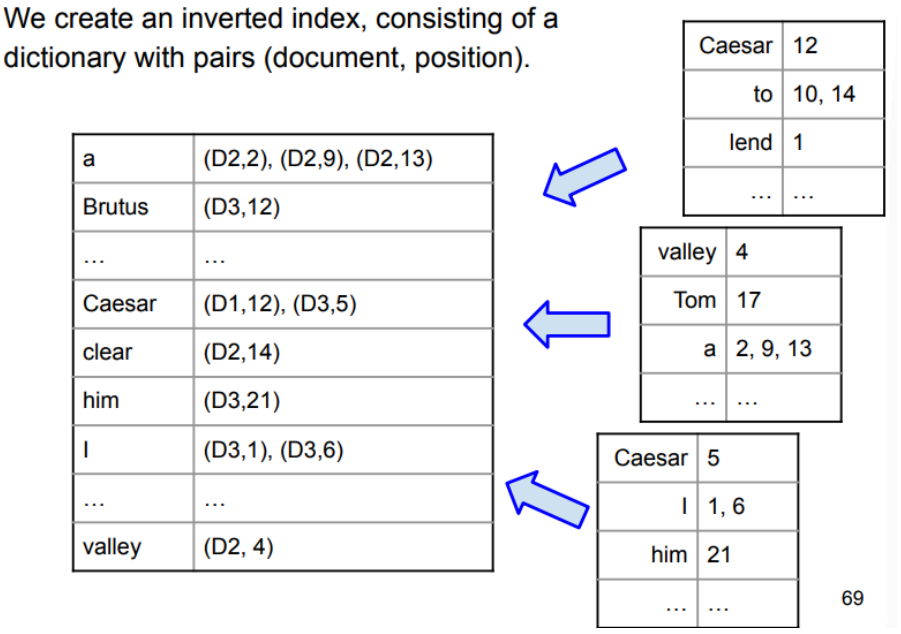
> **Def**
>
> Mappano ogni parola chiave (o termine) ai documenti in cui questa appare.
> Invece di creare un indice per ciascun attributo o parola, viene costruito un indice invertito che rappresenta tutti i documenti in cui un termine specifico è presente.

Gli indici invertiti sono progettati per supportare due tipi principali di query:

1. **Query con set di parole chiave**:
   - *obiettivo*: recupero tutti i documenti che contengono un dato insieme di parole chiave $K_1, K_2, \ldots, K_n$.
   - ad esempio: se cerchiamo i documenti che contengono "data" e "index":
     - l'indice invertito ci fornisce due liste di documenti una associata a "data" e una associata a "index";
     - le liste vengono intersecate per trovare i documenti che contengono entrambe le parole.
2. **Query con sequenza di parole**:
   - *obiettivo*: recuperare tutti i documenti che contengono una sequenza precisa di parole chiave $K_1, K_2, \ldots, K_n$.
   - ad esempio: se cerchiamo "hash table performance":
     - l'indice invertito ci permette di verificare la posizione delle parole nei documenti;
     - restituisce solo i documenti in cui la sequenza delle parole è esattamente quella specificata.

Costruzione di un indice invertito:





Per migliorare l'efficienza e l'accuratezza delle ricerche attraverso l'uso degli indici invertiti abbiamo a disposizione tre tecniche che mirano a:

- velocizzare il recupero delle informazioni: ottimizzando la struttura dell'indice;
- migliorare la precisione dei risultati: eliminando elementi ridondanti o irrilevanti.

Token Normalization:

> **Def**
>
> La normalizzazione dei token consiste nel trasformare le parole in una forma standard, eliminando differenze superficiali tra le sequenze di caratteri.

Esempio:

- la parola "Windows" (con la maiuscola) viene trasformata in "windows" (w minuscola).

- questo permette di considerare equivalenti parole che differiscono solo per maiuscole/minuscole o altri aspetti formattivi.
  Vantaggi:
- **uniformità**: riduce le variazioni superficiali, rendendo le ricerche più precise;
- **efficienza**: migliora il recupero di documenti perché non è necessario gestire forme diverse della stessa parola.

Stemming:

> **Def**
>
> Lo stemming è il processo di riduzione delle parole alla loro radice o "stem", rimuovendo prefissi e suffissi.

Esempio:

- le parole "fishing", "fished" e "fisher" vengono ridotte alla radice comune "fish";
- i sostantivi plurali come "cars" possono essere trasformati nella loro forma singolare "car".
  Vantaggi:
- **riduzione delle varianti**: consente di trattare parole con significati simili come equivalenti, migliorando il recupero di documenti rilevanti.
- **compressione**: riduce il numero di termini nell'indice, rendendolo più compatto.

Stop Words:

> **Def**
>
> Le stop words sono le parole più comuni (come "the", "and", "of"...) che spesso non aggiungono valore semantico alla ricerca e vengono escluse dall'indice.

Motivazione: queste parole compaiono in quasi tutti i documenti, quindi non aiutano a distinguere documenti rilevanti da non rilevanti.
Esempio:
In una query come "find the best car", la parola "the" viene ignorata.
Vantaggi:

- **riduzione della dimensione dell'indice**: eliminando le stop words, l'indice diventa più piccolo e più veloce da interrogare.
- **precisione**: le query producono risultati più mirati eliminando parole poco significative.

Esempi:

Return all the documents that contain both the words “Brutus” and “Caesar”.

a	(D2,2), (D2,9), (D2,13)
Brutus	(D3,12)
…	…
Caesar	(D1,12), (D3,5)
clear	(D2,14)
him	(D3,21)
I	(D3,1), (D3,6)
…	…
valley	(D2, 4)

→ D3

∩ → D3

→ D1, D3

I did enact Julius **Caesar**: I was killed i' the Capitol; **Brutus** killed me. It was a brute part of him to kill so capital a …

Return all the documents that contain the sequence “a valley”.

a	(D2,2), (D2,9), (D2,13)
Brutus	(D3,12)
…	…
Caesar	(D1,12), (D3,5)
clear	(D2,14)
him	(D3,21)
I	(D3,1), (D3,6)
…	…
valley	(D2, 4)

→ (D2,2), (D2,9), (D2,13)

∩ → D2: ~~2, 4~~ ~~9, 4~~ ~~13, 4~~

→ (D2,4)

Return all the documents that contain the sequence “a clear”.

a	(D2,2), (D2,9), (D2,13)
Brutus	(D3,12)
…	…
Caesar	(D1,12), (D3,5)
clear	(D2,14)
him	(D3,21)
I	(D3,1), (D3,6)
…	…
valley	(D2, 4)

→ (D2,2), (D2,9), (D2,13)

∩ → D2: 2, 14 / 9, 14 / 13, 14

→ (D2,14)

In a broad valley, at the foot of a sloping hillside, beside **a clear** bubbling stream, Tom was building …