Hashing Techniques for Secondary Storage

Important: Must minimize number of disk accesses.

- Assume a page holds b records.

  Obvious solution:

  Hash to an array of m pages (each holds b records) ... Chain overflows.

  This works well but

  – A few elements will be at end of long chains - not good.

  – n grows, chains too long

- We want a method which would guarantee all records can be retrieved in a couple of disk accesses.

Extensivle Hashing

- Every record can be retrieved in 2 disk accesses

- Cost: We must keep an index on disk. Size $\geq \frac{2n}{b}$

- like B-tree: when a page is full, split it into two.

- Hash function: (1) Key mod p, $p \geq$ max file size.
  (2) We will use the following function:
  Key $\Rightarrow$ long pseudo random bit string ("signature")

  - first few bits will be used

  - but we do not know "how many" in advance.

We map records to pages - no page contain $> b$ records.

The Idea. At any time, we know how many bits of hash sequence are needed so that no more than b records have same "prefix".

Let $d = |prefix|$.

Keep 2^d pointers to appropriate locations. Not all need be distinct.

Let's look at the next example.

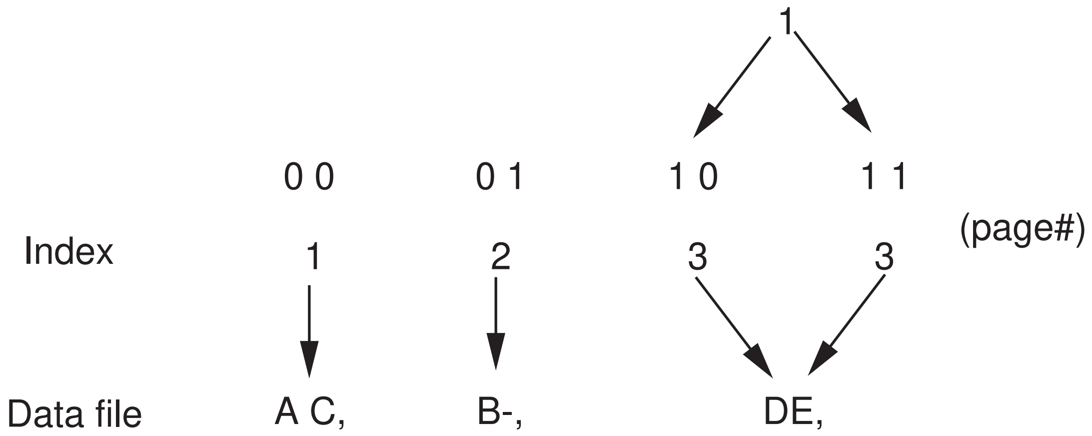
Example.

Let $b = 2$. i.e. 2 records per page.

Assume we have mapping

$$
\begin{array}{lccccccc}
A & \rightarrow & 0 & 0 & 0 & 0 & 1 & \ldots \\
B & \rightarrow & 0 & 1 & 0 & 0 & 1 & \ldots \\
C & \rightarrow & 0 & 0 & 1 & 1 & 0 & \ldots \\
D & \rightarrow & 1 & 0 & 1 & 0 & 1 & \ldots \\
E & \rightarrow & 1 & 1 & 0 & 1 & 0 & \ldots
\end{array}
$$

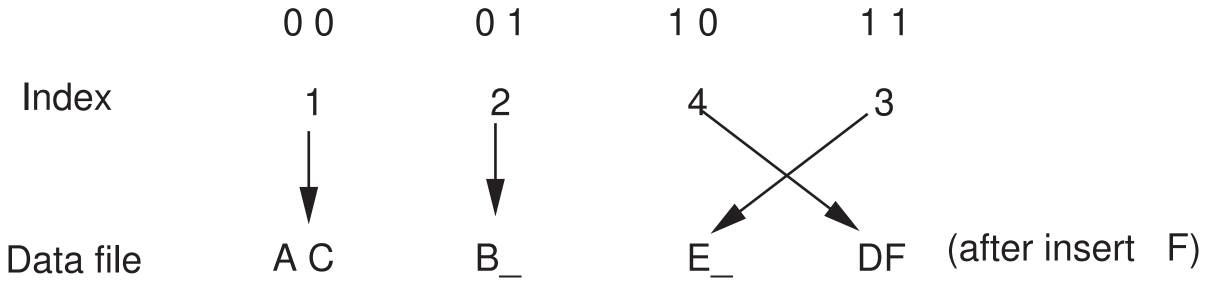
$d = 2$



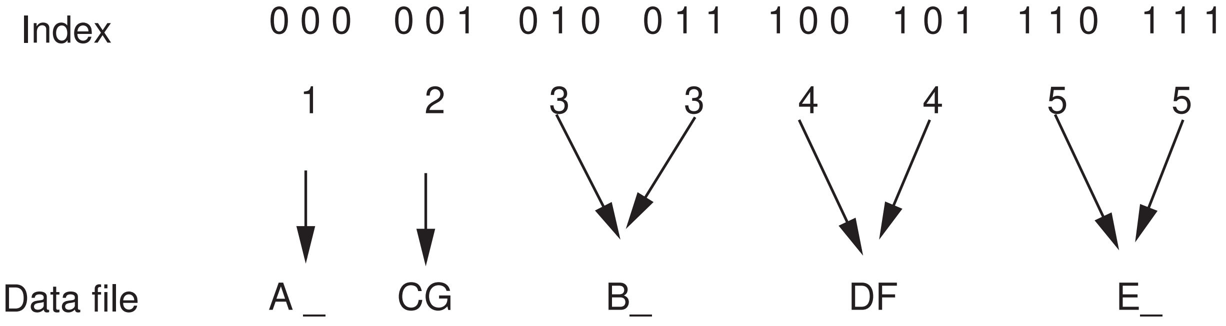
- Note that page 3 contains all records with signature of form "1-". (page 3's prefix length is only 1).

Example (cont'd): Now add F. Assume F $\rightarrow$ 10010. Bucket 3 must split.

A	$\rightarrow$	0	0	0	0	1	-	-	
B	$\rightarrow$	0	1	0	0	1	-	-	
C	$\rightarrow$	0	0	1	1	0	-	-	
D	$\rightarrow$	1	0	1	0	1	-	-	
E	$\rightarrow$	1	1	0	1	0	-	-	-
F	$\rightarrow$	1	0	0	1	0	-	-	-



Insert G $\rightarrow$ 0 0 1 0 1



In summary, if we add a record: either it fits in (no overflow), or the page overflows and must be split. When we have "overflow", there are 2 cases:

$k < d$: Split according to next bit, creating a new data page. This will also cause a number (2^{d-k}) of pointers to be altered in the index table.

$k = d$: Split according to next bit. This means $d + 1$ bits of the prefix are needed. Hence all 2^d pointers in index file must be duplicated, then a few changed.

Note: this happens rarely. But when this happens, the file doubles in size.