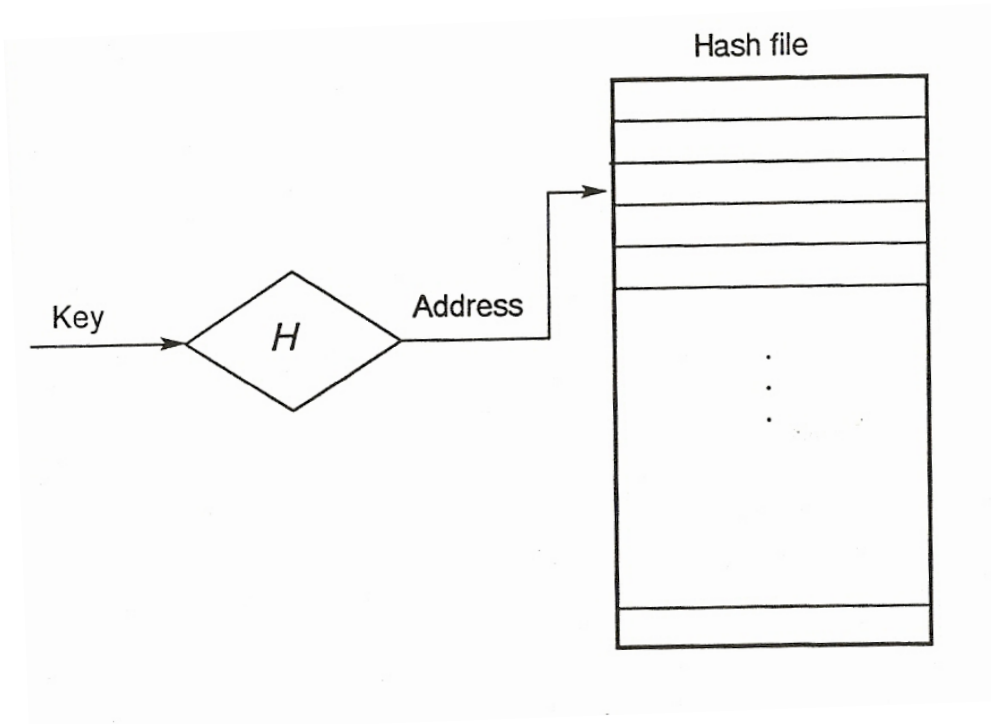


Basic File Structures – The Hash (Direct) File (1)

Hashing (**hash addressing**) is a technique for providing fast **direct** access to a specific record on the basis of a given value of some field. The disk address (usually, address of a page) is computed from the value of the field by a **hash function H** :



If two or more key values hash to the same disk address, we have a **collision**. The hash function should distribute the domain of the key possibly evenly among the address space of the file to minimize the chance of collision.

The collisions may cause a page to overflow. In this case:

- overflow pages may be used (**chaining**), or
- searching for next page with free space (**open addressing**) may be performed.

This “traditional” hash file organization is similar to that applied in hash tables in main memory.

In a dynamic file (i.e. in the presence of insertions and/or deletions), a hashed file must be periodically reorganized.

Static hashing (1)

The average number of disk accesses necessary to fetch a record is $1+p$, where p denotes additional cost due to collisions. If we assume the blocking factor equal to 1 (i.e. one record per disk page), then the file structure is the same as for hashed tables. If m denotes file capacity (in records), n - number of records inserted, then the average number of disk accesses is

$$S^+ = 1 + 0.5 * \frac{n}{m}$$

for separate chaining, and

$$S^+ = 1 + 0.5 * \frac{n}{m-n}$$

for open addressing with linear search.

If the blocking factor is greater than 1 (as it is usually in real-life), the situation for separate chaining does not change much, as in a large file and at random insertions, the probability that the elements of the list of synonyms are on the same page is low. In contrast, for linear search, the probability that colliding keys are on the same page is high, as the collision resolution method tries to locate a colliding key as close as possible to its proper (i.e. calculated by the hashing function) place. The effect of clustering is advantageous! (Compare with hashing in main memory!).

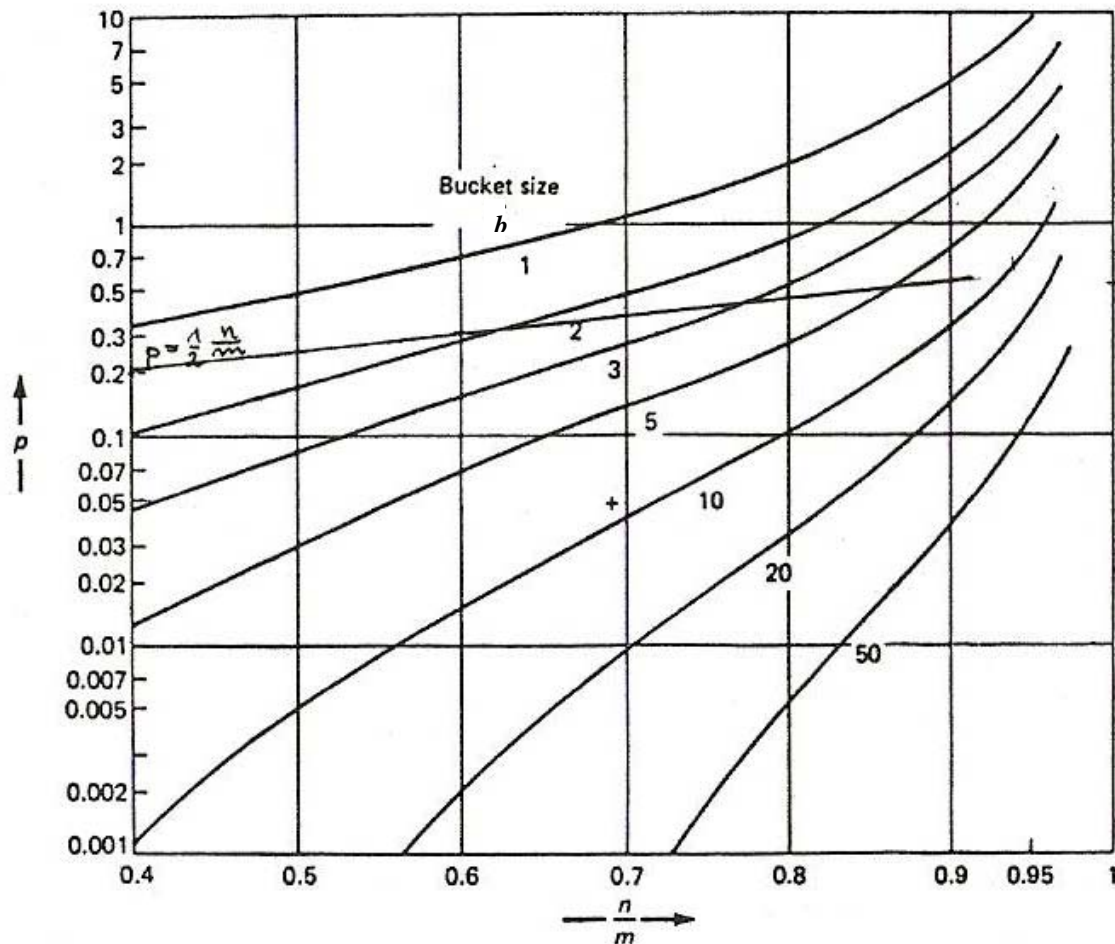
Note that in this case use of other (quadratic or cubic) search methods is not appropriate, as they try to spread the keys over the entire table, and the probability that colliding keys are on the same page is low.

Static hashing (2)

The following chart depicts the collision cost p of open addressing with linear search as a function of file density $a = n/m$. Multirecord pages (buckets) with blocking factor b are assumed.

For comparison purposes, also the line for separate chaining is indicated.

From the chart it is seen, that if the blocking factor exceeds 10, then efficiency of hashing based on open addressing with linear search is very good even at high file densities ($\cong 0.95$), and is much better than separate chaining.



Static hashing - an example

Assume that $m = 1.5n$, i.e. $n/m = 0.666\dots$

A. Blocking factor $b = 1$

1. Separate chaining:

$$p = n/2m = 0.333\dots \quad (33\% \text{ additional accesses})$$

2. Open addressing with linear search:

$$p = n/2(m-n) = 1.0 \quad (1 \text{ additional access})$$

To be more fair for open addressing, we should add to m the expected size of overflow area o that is used by separate chaining. A reasonable estimate is $o = 0.5n$, which yields $m = (1.5+0.5)n = 2n$, and

$$p = 0.5 \quad (50\% \text{ additional accesses})$$

B. Blocking factor $b = 10$.

Assume again $m = 1.5n$ ($n/m = 0.666\dots$). For open addressing with linear search, we obtain from the chart on the previous page

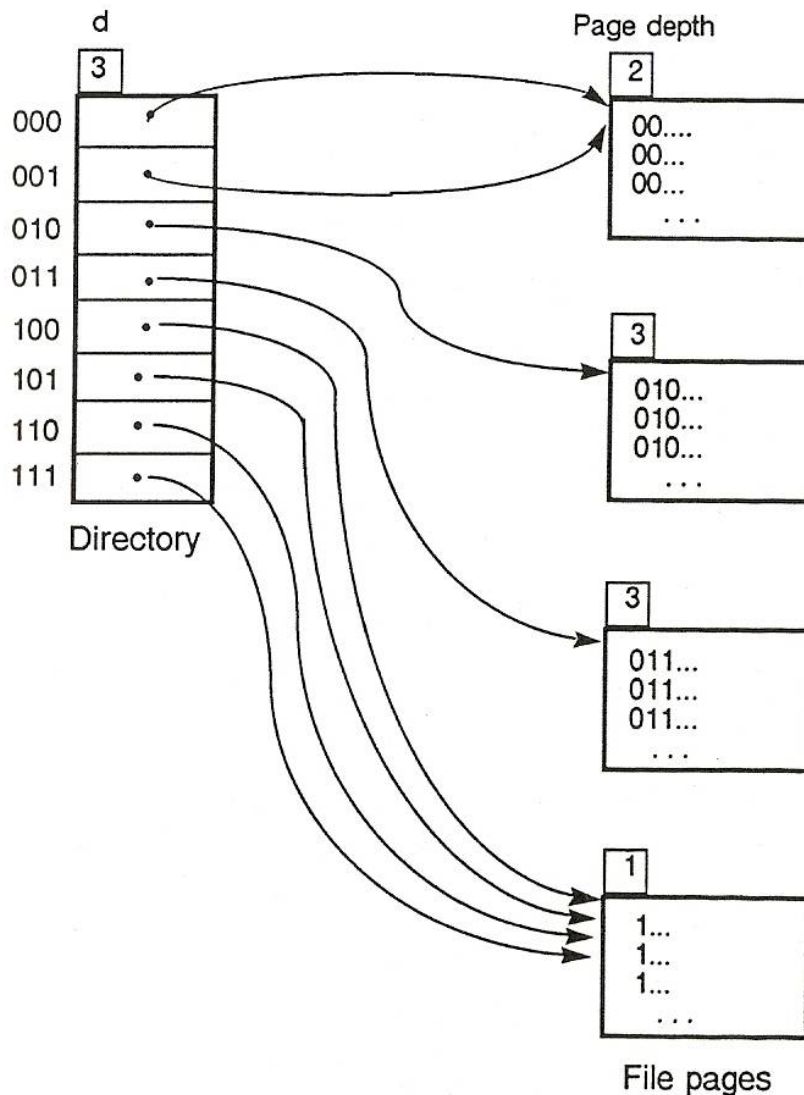
$$p = 0.03$$

which practically ensures one disk access per one fetch-record operation.

Extendable hashing

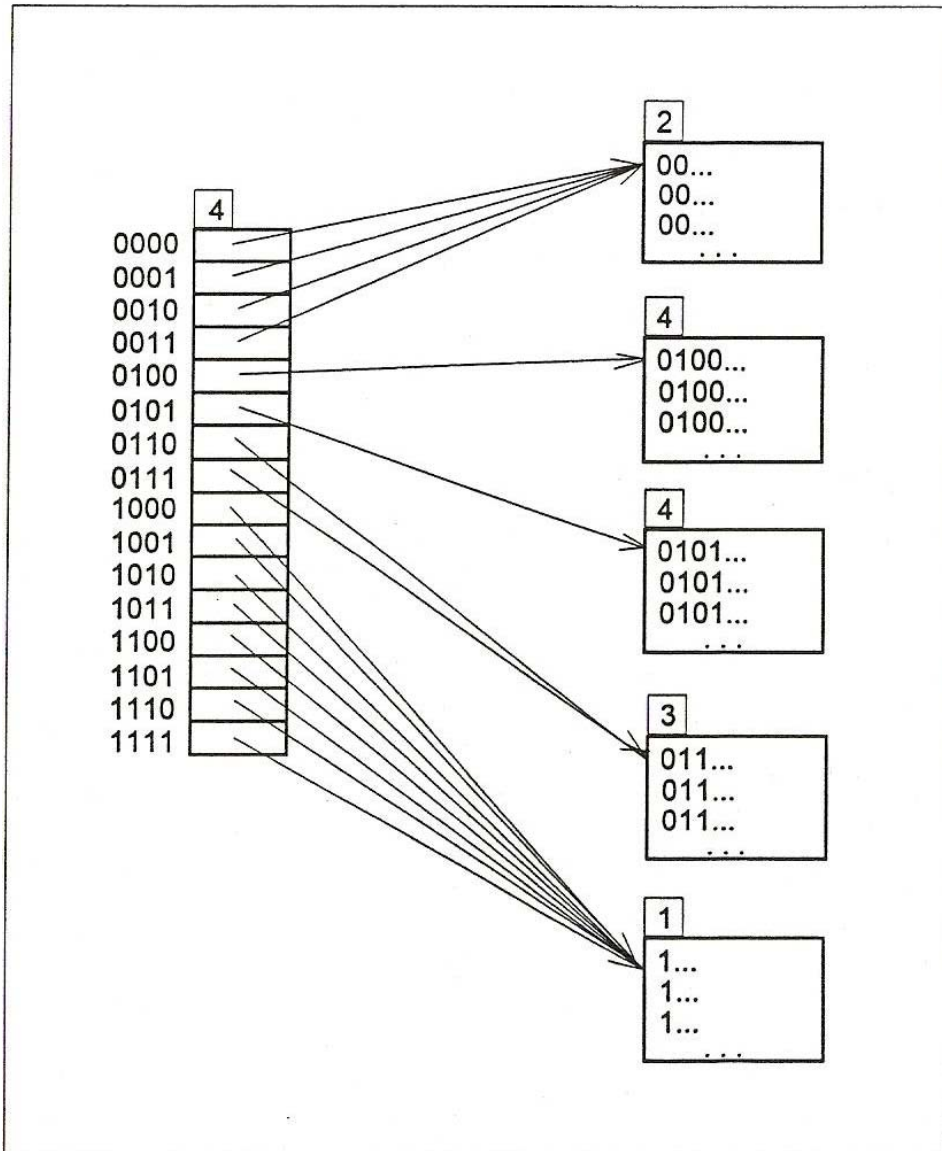
To alleviate the problem of reorganization of a hash file, some dynamic organizations have been invented.

In **extendable hashing**, there is a directory (kept as a file or, if small, in main memory) that keeps pointers to file pages. The position in the directory for a given key is established by the first d bits of the value of hash function ("pseudokey"). The value of d is called the depth of the directory. If a page overflows, it is split into two pages and the records are distributed among the two pages. The directory is modified appropriately, with possibly increasing the current depth by one



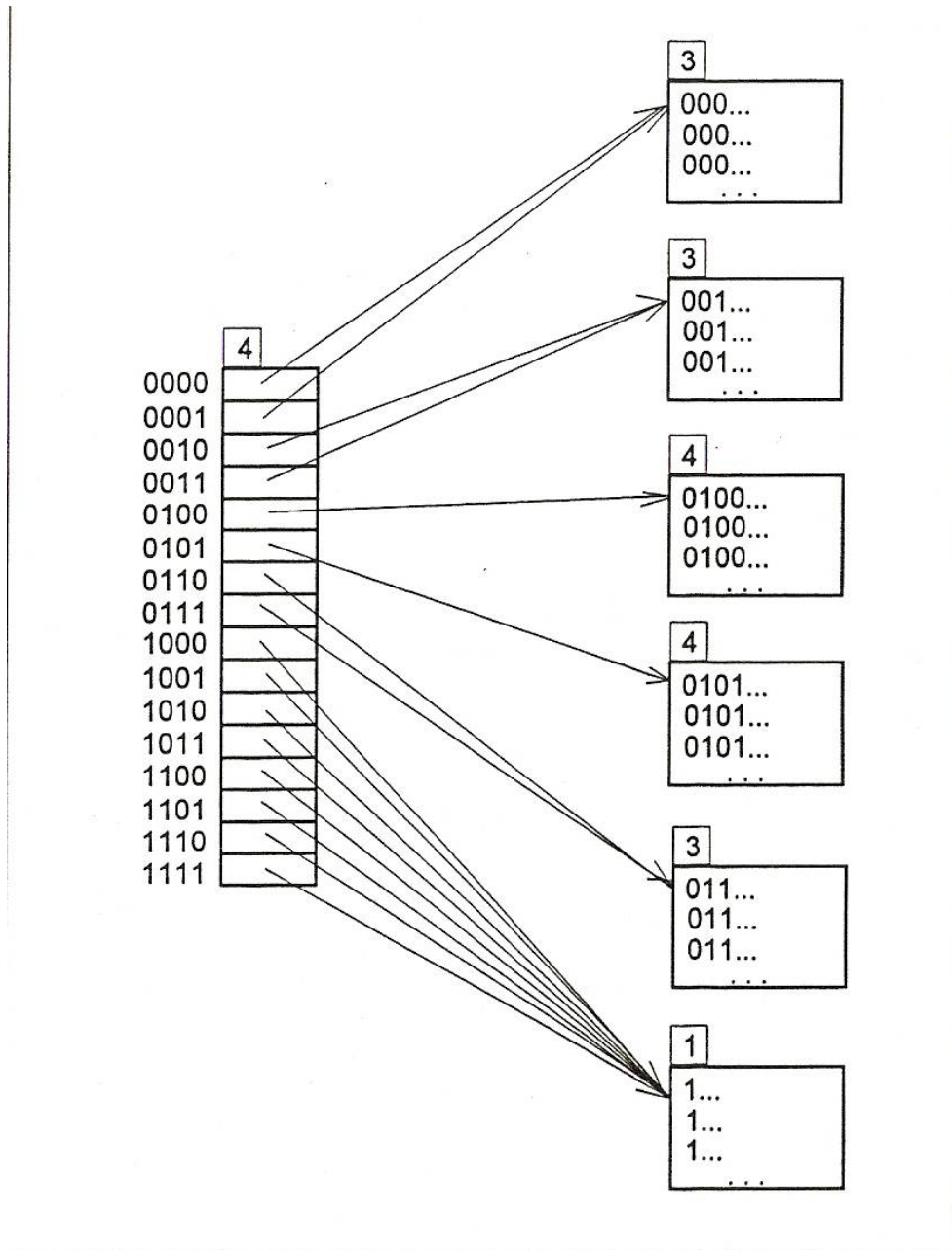
Extendable hashing (ctd.)

Assume that the page No 2 (with the prefix 010) overflows and is split into two pages. As the depth of the page is the same as the depth of the directory, the directory must be doubled, with appropriate updating of pointers.



Extendable hashing (ctd.)

If now the page No 1 (prefix 00) overflows, there is no need to increase the depth of the directory, as the depth of the overflow page does not exceed the depth of the directory.



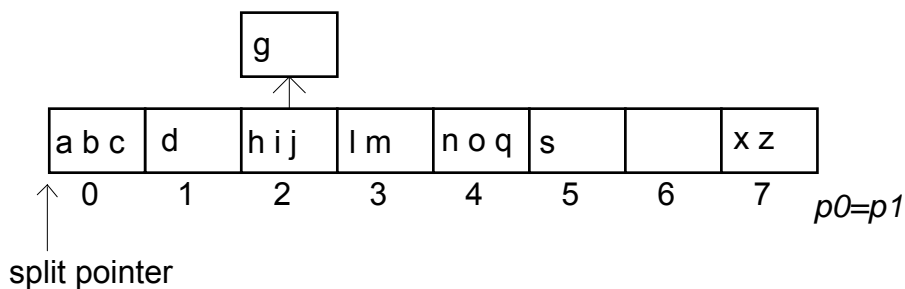
Linear hashing (1)

Linear hashing can be viewed on as a dynamic version of hashing with separate chaining: the overflows are chained together in a separate overflow area, but the size of the file grows (or shrinks) incrementally, page by page, keeping the file density approximately constant.

In the following illustrations, records are represented by letters a, b, c, ..., the blocking factor $b = 3$ (i.e. three records per page), the file density is $\alpha = n/m = 2/3$, where n is the number of records in the file, m is the file capacity (in records).

1. Initial situation:

basic allocation (always a power of 2) is $p0 = 8$ pages (file depth $d = 3$),
current allocation $p1$ equal to basic allocation,
 $n = 16$ records in the file, ($\alpha = 16/24 = 2/3$).



The hash function values (pseudokeys) of records a, b, c have suffix (i.e. low order bits) 000, for record d - suffix 001, for records g, h, i, j - suffix 010, etc. The suffix determines the address of page where the record is stored. The length of the suffix is equal to the file depth.

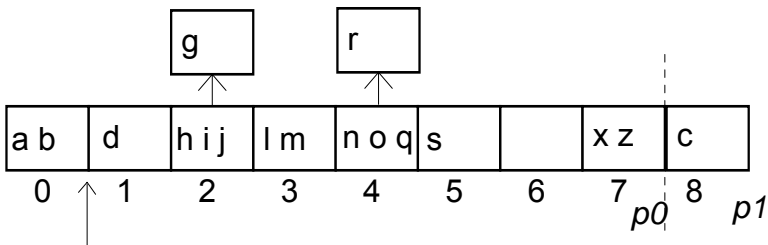
Page No 2 has already overflowed.

The "split pointer" points at the next page to be split.

The file grows from $p1$ to $p1+1$ pages every $\alpha \cdot b = (2/3) \cdot 3 = 2$ insertions.

Linear hashing (2)

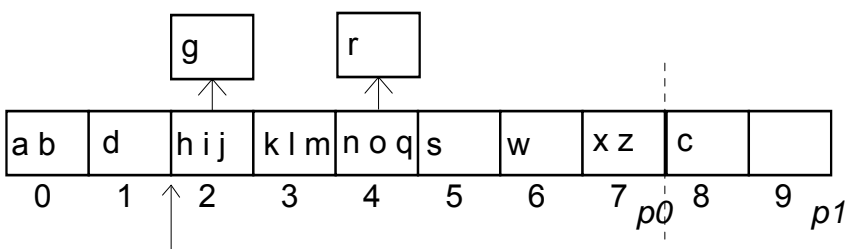
2. After record r with pseudokey suffix 0100 has been inserted:



As the file density would exceed $2/3$, so:

- the new page is allocated to the file,
- the page pointed at by the split pointer (the page with address $p1-p0$) is split,
- current allocation $p1$ increases by one: $p1 = 9$,
- the file depth increases by one,
- the records from the split page (a, b, c) are rehashed with the pseudokey extended by one bit to the left: records a, b, that have suffix 0000, remain on page 0; record c, that has suffix 1000, is moved to the newly allocated page 8.

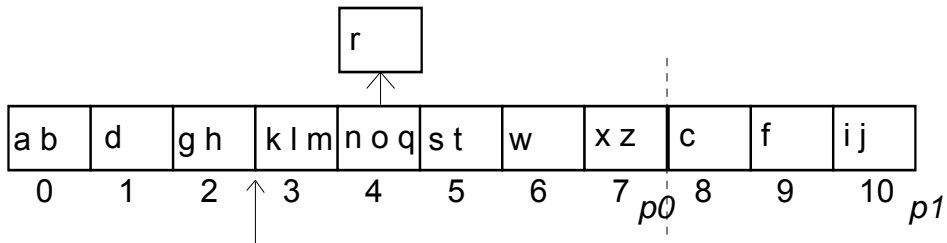
3. After next 2 insertions: record k (suffix 0011) and w (suffix 0110), the next page (address 1) is split.



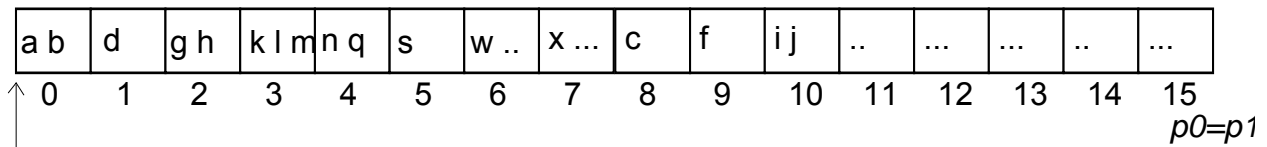
The record d has the pseudokey suffix 0001, so it remained on page 1. In this case the newly allocated page 9 is empty at the moment.

Linear hashing (3)

4. After next 2 insertions: record f (suffix 1001) and t (suffix 0101), the next page (address 2) together with its overflow page is split.



The process continues until $p1$ becomes $2 \cdot p0$. Then, the whole process starts anew with the value of $p0$ doubled and the file depth increased by one:



Note that the split pointer points at page 0 again.

Linear hashing (4)

For the linear hashing scheme to work properly, the hashing function KAT must dynamically modified according to the current file depth d . For a given key k , KAT produces an address in the range $0..2 \cdot p_0 - 1$, so that always d bits are available. Then, the generated address is reduced to the current range $0..p_1 - 1$:

```
address := KATd(k)
IF address > p1-1 THEN
    address := address - p0
```

The records from the split page (possibly - with overflow pages) have either the original address ("0" on the extra bit) and they stay in their home page, or have the new address p_1 ("1" on the extra bit), referring to the new page.

When the file has grown to twice its size ($p_1 = 2 \cdot p_0$), KAT is modified to produce another bit with the low order bits remaining the same, the values of p_0 and p_1 are doubled and the process continues starting with $p_1 = p_0$.

If there are deletions, the whole procedure can work the other way round, with merging appropriate pages instead of splitting.

Performance of linear hashing may be estimated in a similar way as for separate chaining. The cost of accessing a record is 1 access (address calculated by KAT) plus expected cost of traversing the overflow chain.

$$S^+ = 1 + \alpha/2$$

Indeksy bitowe (1)

Idea:

Wykorzystanie pojedynczych bitów do zapamiętania informacji o tym, że dana wartość atrybutu występuje w określonej krotce relacji.

Dla każdej wartości atrybutu w przechowywana jest tablica bitów („mapa bitowa”) $A(w)$, której każdy bit odpowiada jednej krotce relacji:

$A(w)[n] = 1 \Leftrightarrow$ atrybut A krotki nr n przyjmuje wartość w ,
 $A(w)[n] = 0$ w przeciwnym razie.

Liczba bitów w każdej mapie bitowej jest równa liczbie krotek indeksowanej relacji.

	w_1	w_2	w_3	w_4	...	Wartości atrybutu
1	0	0	1	0		
2	1	0	0	0		
3	1	0	0	0		
4	0	1	0	0		
5	1	0	0	0		
6	0	0	0	1		
7	1	0	0	0		
8	1	0	0	0		
9	0	1	0	0		
10	0	0	0	1		
·	·	·	·	·		
·	·	·	·	·		
·	·	·	·	·		

Indeksy bitowe (2)

Przykład:

Indeksowana relacja:

Sprzedaż		
KlientID	Marka	Kolor
1010	Ford	zielony
1020	BMW	niebieski
1030	Ford	zielony
1040	Audi	czarny
1050	Volvo	zielony
1060	Fiat	niebieski
1070	Ford	niebieski
1080	Opel	zielony
1090	Opel	niebieski
1100	Ford	czarny

Indeks bitowy:

Kolor		
Kolor(zielony)	Kolor(niebieski)	Kolor(czarny)
1	0	0
0	1	0
1	0	0
0	0	1
1	0	0
0	1	0
0	1	0
1	0	0
0	1	0
0	0	1

Indeksy bitowe (3)

Zalety:

- Mały rozmiar dla atrybutów o wąskiej dziedzinie wartości (np. jeśli relacja ma 1 000 000 krotek, a atrybut przyjmuje tylko 4 różne wartości, to łączny rozmiar indeksu wyniesie $4 \cdot 125 \text{ kB} = 0,5 \text{ MB}$; natomiast rozmiar B-drzewa będzie nie mniejszy niż 4 MB).
- Duża szybkość przetwarzania zapytań określonych klas: z zastosowaniem operatorów logicznych AND, OR i NOT oraz zapytań z operatorem COUNT.

Wady:

- Duży rozmiar dla atrybutów o licznej dziedzinie wartości (np. jeśli atrybut przyjmuje 256 różnych wartości, to rozmiar indeksu bitowego o 1 000 000 krotek wyniesie 32 MB i będzie wiele razy większy niż rozmiar odpowiedniego indeksu opartego na B-drzewie). W efekcie nie mogą być stosowane jako indeksy główne.
- Mniejsza efektywność przy zapytaniach zakresowych (z użyciem operatorów $<$, $>$ itp.) i innych, przy których trzeba dokonywać operacji bitowych na wielu (lub wszystkich) mapach bitowych.
- Mniejsza elastyczność (np. w operacjach usuwania).

Zastosowanie:

- Hurtownie danych
- Inne bazy danych, w których atrybuty przyjmują wartości z bardzo ograniczonych liczebnie dziedzin

Stosowane np. w DBMS: Oracle, Informix, Sybase, RedBrick, ...

Indeksy przestrzenne

Motywacje:

- Wiele aplikacji (CAD, GIS, aplikacje multimedialne) operują na *danych przestrzennych*, tj. takich, które są skojarzone ze współrzędnymi przestrzennymi, z obszarami 2- i 3-wymiarowymi.
Danymi przestrzennymi są np.
 - punkty,
 - linie,
 - wielokąty,
 - bryły.
- Konwencjonalne systemy baz danych nie potrafią efektywnie przetwarzać danych przestrzennych, bo:
 - ◆ Dane przestrzenne są bardzo obszerne, mają bardzo złożoną strukturę, pozostają ze sobą w różnych złożonych zależnościach.
 - ◆ Proces wydobywania informacji z bazy danych bazuje na operatorach niespotykanych w klasycznych systemach przetwarzania danych:
 - przecięcie (*intersection*),
 - nakładanie się (*overlapping*)
 - sąsiedztwo (*adjacency*),
 - bliskość (*proximity*),
 - zawieranie się (*containment*),
 - ...
 - ◆ Trudno jest zdefiniować porządek pomiędzy obiektami przestrzennymi (klucze), w związku z czym nie można stosować tradycyjnych technik indeksowania

Potrzebne są nowe techniki indeksowania

Indeksy przestrzenne – R-drzewo (Guttman)

R-drzewa (różne warianty) stosowane są w modułach rozszerzających relacyjne DBMS o możliwości przetwarzania danych przestrzennych:

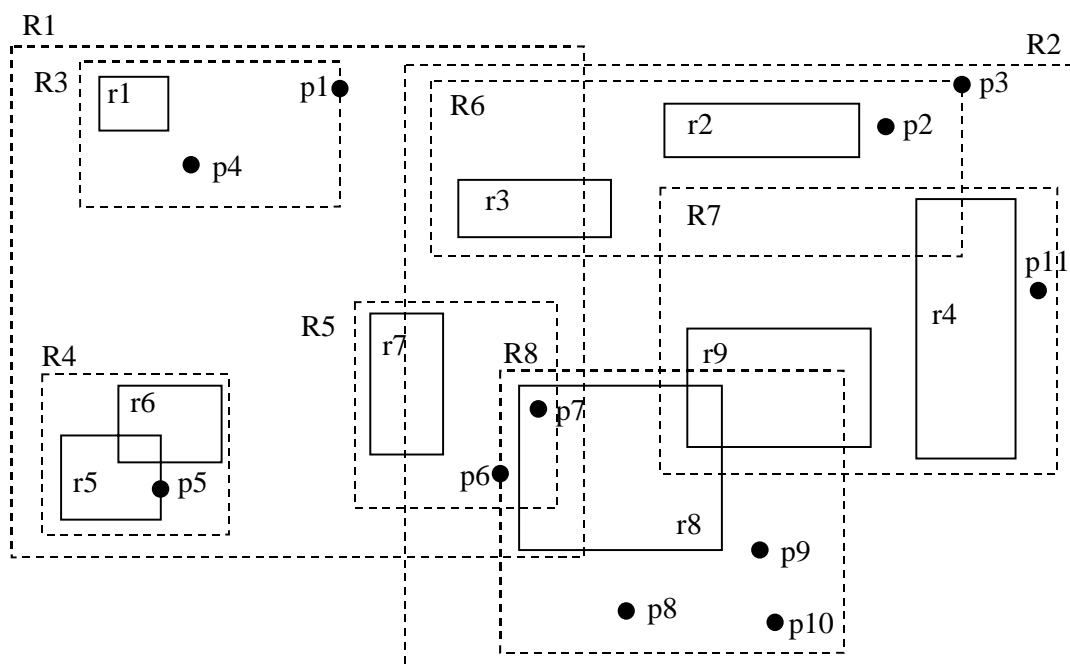
Dynamic Server with Universal Data Option (Informix)

Oracle (InterMedia Cartridge)

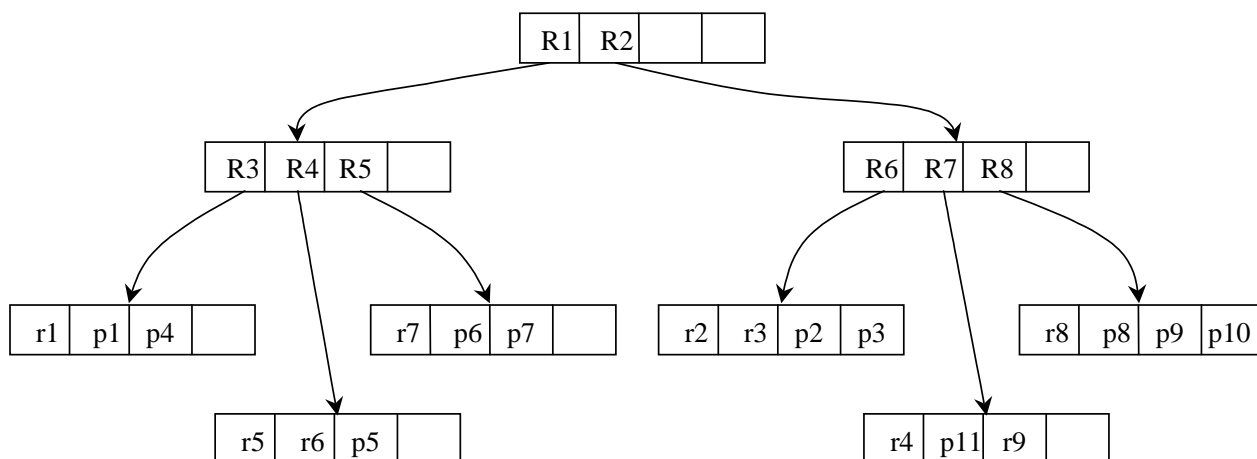
DB2 Universal Database (Spatial Extender)

W R-drzewie prostokąty ograniczające mogą się nakładać.

Przykładowy układ danych przestrzennych:



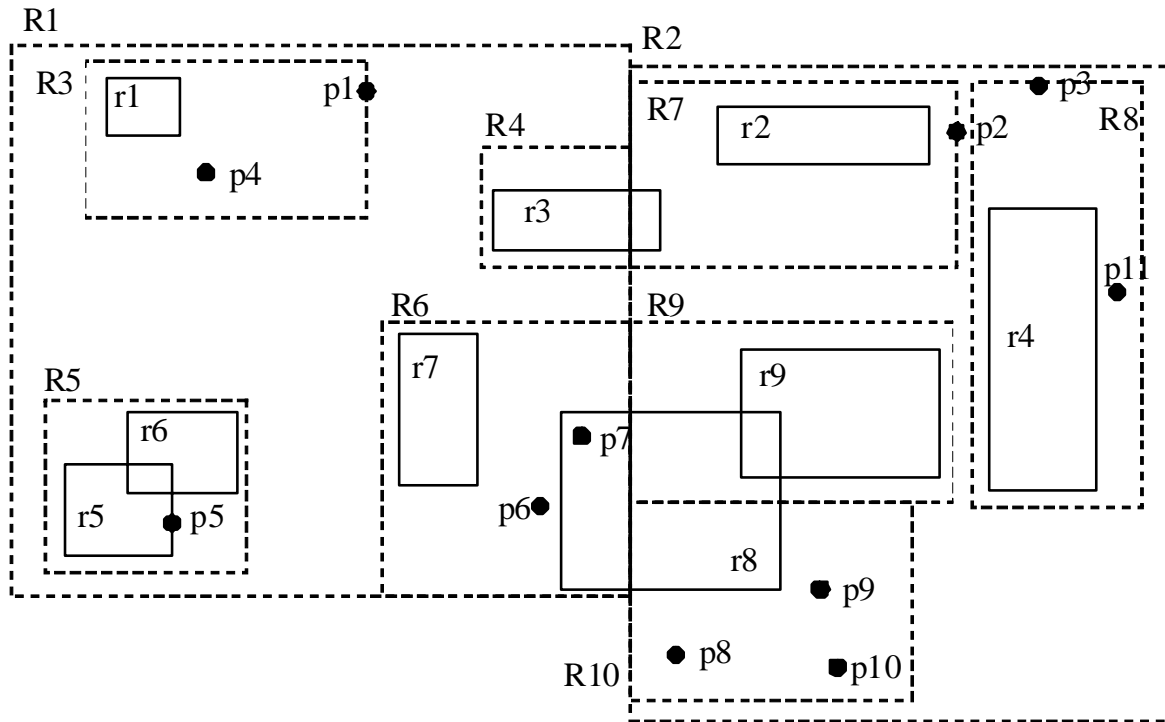
Odpowiadające mu R-drzewo:



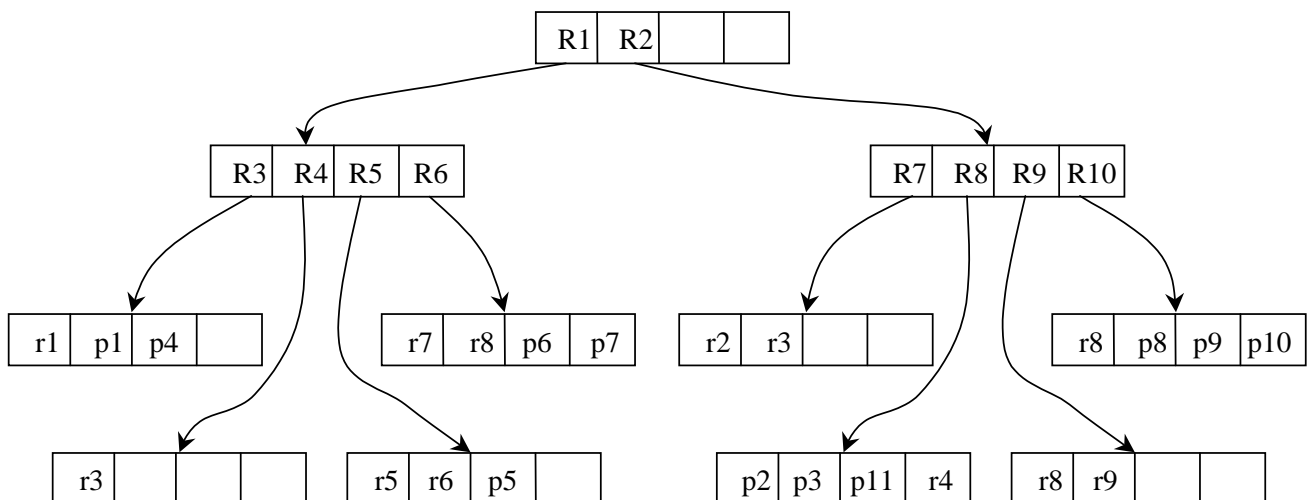
Indeksy przestrzenne – R⁺-drzewo (Sellis)

W R⁺-drzewie prostokąty ograniczające są rozłączne.

Przykładowy układ danych przestrzennych:

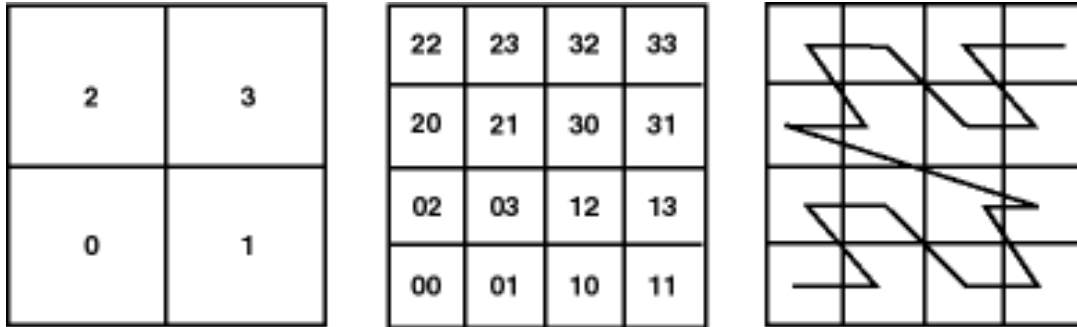


Odpowiadające mu R⁺-drzewo:



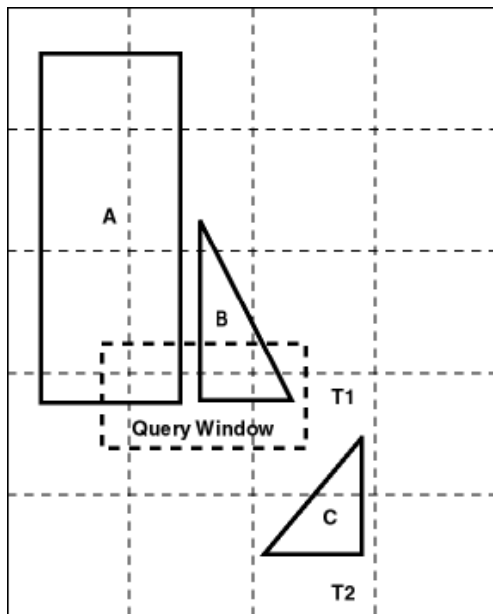
Indeksy przestrzenne – Quadtree

Przestrzeń (zazwyczaj – planarna) jest dzielona na kwadratowe „kafelki” (*tiles*). Kafelki są ponumerowane liniowo.

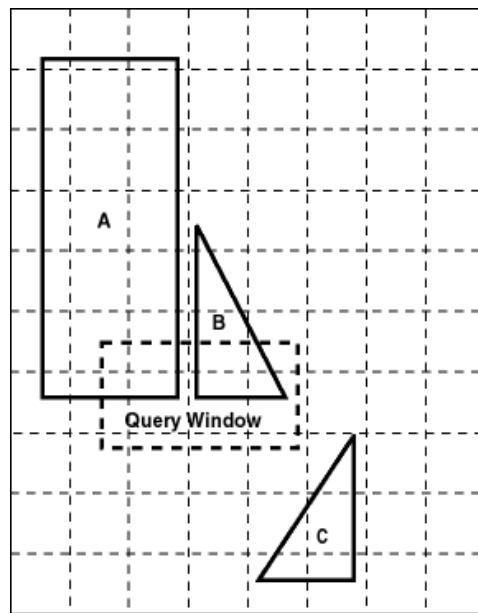


Im więcej kafelków (im wyższy poziom kafelkowania), tym precyzyjniej można określić położenie figury w przestrzeni.

1.

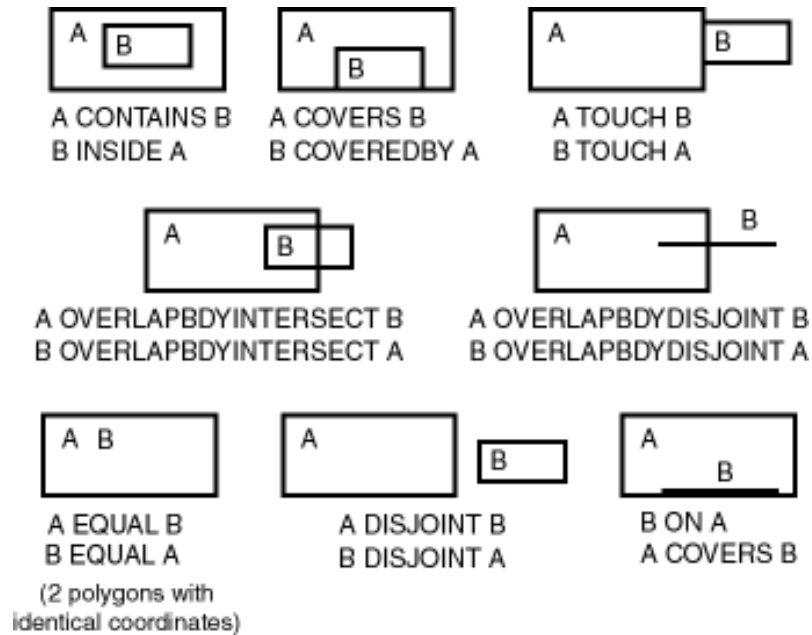


2.



Na rys. 1. figury B i C zajmują ten sam kafelek, a więc potencjalnie kolidują ze sobą. Z rys. 2 wiadomo, że figury te nie kolidują, gdyż nie mają wspólnego kafełka.

Relacje przestrzenne figur



Relacje przestrzenne pomiędzy figurami zależą od relacji pomiędzy ich wnętrzami i brzegami.

- CONTAINS** - wnętrze i brzeg jednej figury są całkowicie zawarte we wnętrzu drugiej figury
- INSIDE** - przeciwne do CONTAINS: $A \text{ INSIDE } B \Leftrightarrow B \text{ CONTAINS } A$
- COVERS** - wnętrze jednej figury jest całkowicie zawarte we wnętrzu drugiej figury i ich brzegi przecinają się
- COVEREDBY** - wnętrze jednej figury jest całkowicie zawarte we wnętrzu drugiej figury i ich brzegi przecinają się
- COVEREDBY** - przeciwne do COVERS: $A \text{ COVEREDBY } B \Leftrightarrow B \text{ COVERS } A$
- TOUCH** - brzegi przecinają się, ale wnętrza nie przecinają się
- OVERLAPBDYINTERSECT** - granice i brzegi obu figur przecinają się
- OVERLAPBDYDISJOINT** - wnętrze jednej figury przecina brzeg i wnętrze drugiej figury, ale oba brzegi nie przecinają się
- EQUAL** - obie figury mają to samo wnętrze i brzegi
- DISJOINT** - brzegi i wnętrza nie przecinają się
- ON** - wnętrze i brzeg jednej figury leżą na brzegu drugiej figury

Storage Devices

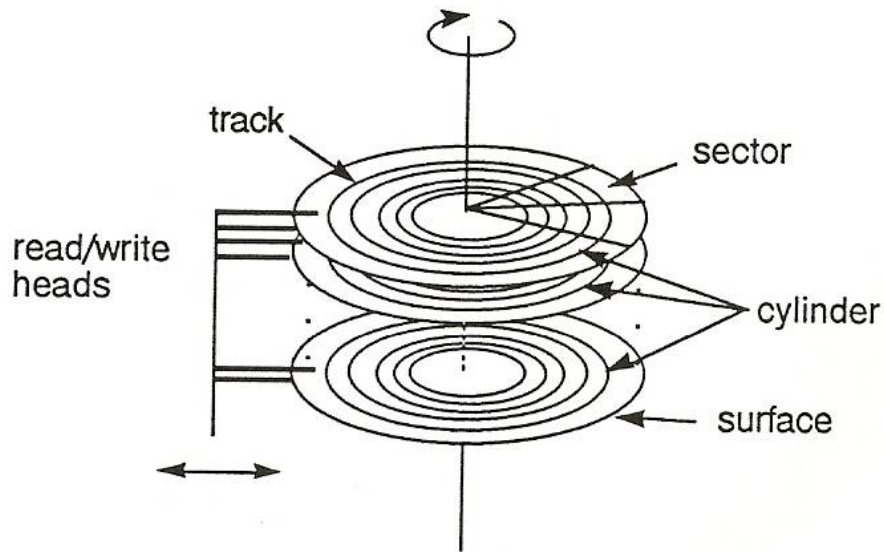
- Internal (physical) level of a database system comprises:
 - external devices
 - file structures and access methods
- External devices
 - magnetic for on-line data access
 - optical disks, streamers for off-line data access (back-ups, archives, safety copies etc.).

Disks provide **random access** to data (so they are called **random access devices**). Random access means that the time needed to access a given piece of data does not depend on the physical placement of the piece of data accessed previously. In other words, access time does not depend on the physical distance between data. RAMs and ROMs memories are examples of genuine random access devices.

Disk memories are not ideal random access devices due to their time characteristics (seek time, rotational delay). However, they are treated as if they provided the real random access in order to distinguish them from **sequential access devices**, for which the access time is greater by several orders of magnitude.

In sequential access devices, the time required to access a piece of data strongly depends on the physical placement of previously accessed data (i.e. on the position of the read/write mechanism). Magnetic tapes of different types are examples of sequential access devices.

Disk Storage (1)



Access time = seek time + rotational delay + transfer time

Seek time t_s – time required to position the heads over the proper track:

$$t_s = a + \hat{\alpha}d$$

where d is the distance between tracks (*seek distance*).

In modern disks, $t_{s,avg} \cong 10$ ms (in optical disks is much greater because optical read/write heads are more massive than in magnetic disks).

Disk Storage (2)

Rotational delay t_r – delay between the completion of seek and the actual transfer of data.

Usually, the average rotational delay equals to the half of the time required for one rotation of the disk.

$$t_{r\text{avg}} = 0.5 \text{ time/disk revolution}$$

In modern disks, $t_{r\text{avg}}$ takes several milliseconds (still the “eternity” as far as RAM access time is concerned).

Transfer time t_t – time required to transfer a block of data.

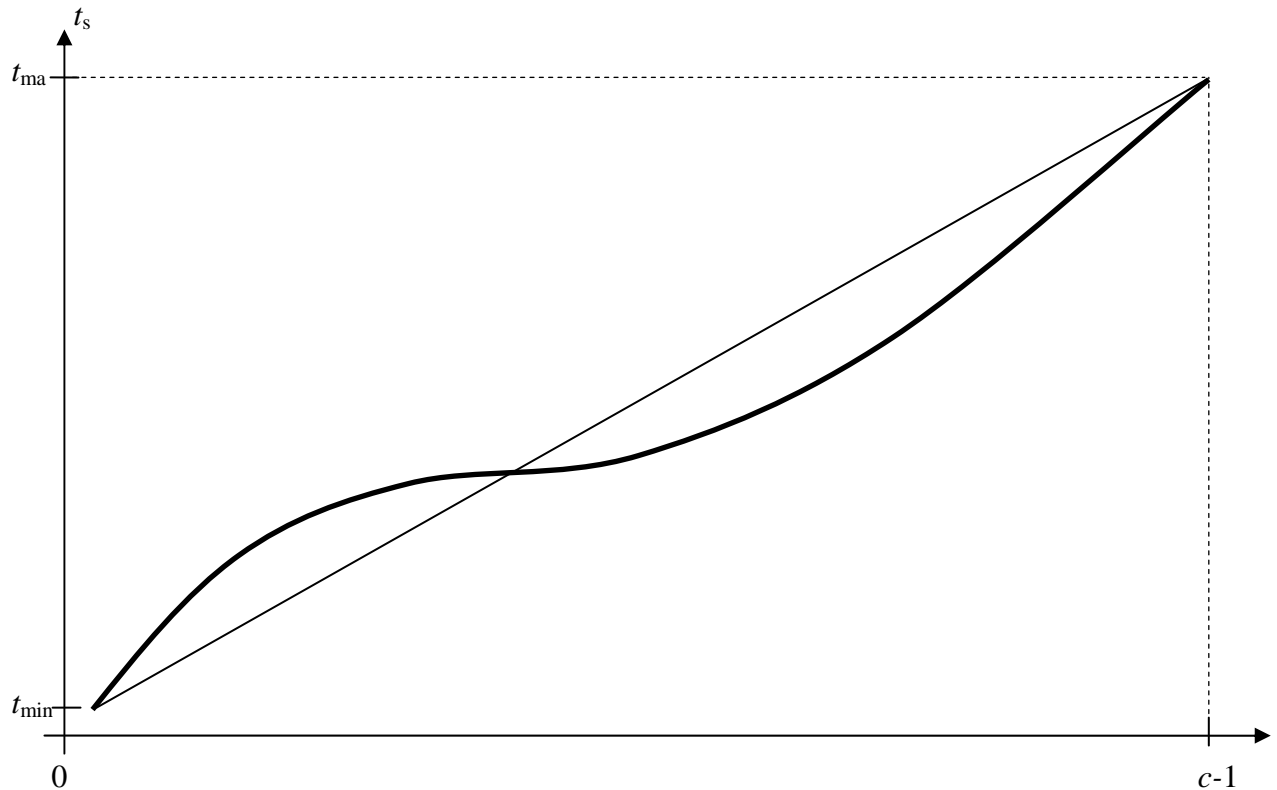
$$t_t = B / r$$

where r is a transfer rate (usually several MBs), and B is the number of bytes to be read or written.

Usually, only a whole number of physical blocks (sectors) may be transferred by one disk transmission.

For efficiency reasons, transmissions between disk and main memory are made in units called *blocks* or *pages*. The fetched disk pages are buffered in main memory so that not each disk transfer request issued from a program is accompanied by a physical transmission of data. An operating system is responsible for appropriately managing the pool of disk buffers.

Disk storage (3)



Typical seek-time characteristics $t_s = f(d)$

t_{\min} - track-to-track seek time

t_{\max} - seek time for $d = c-1$, where c is the number of cylinders

t_s - seek time

d - seek distance

If the starting and ending cylinders are equally probable ("random case"), it can be shown that the expected number of cylinders travelled during a single seek is approx. $c/3$. Hence, as the seek time as a function of seek distance can be approximated with negligible error by a straight line, the average seek times equals

$$t_{s \text{ avg}} \approx t_{\min} + \frac{t_{\max} - t_{\min}}{3}$$

Achieving High Reliability and High Performance

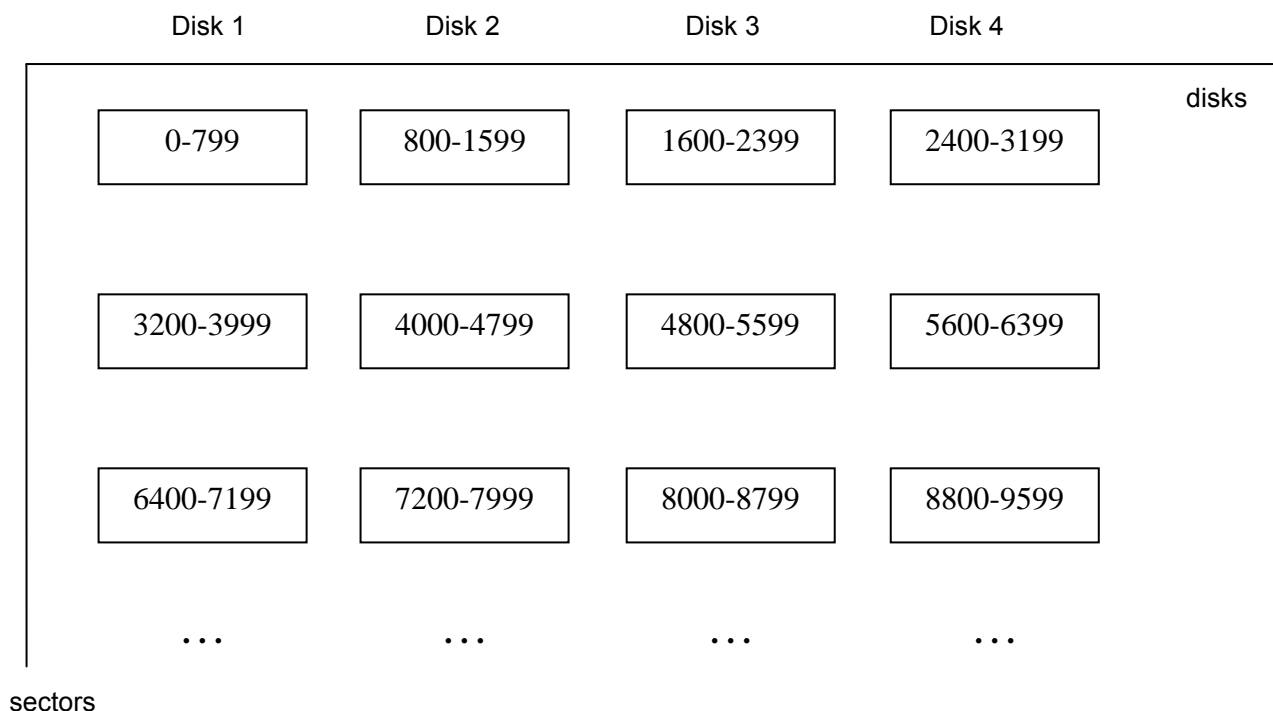
Disk Array storage technology

Idea: replacing a large, expensive disk by a group of smaller and cheaper disks (i.e. by an array of disks). The idea originates from the University of Berkeley.

RAID - Redundant Array of Inexpensive (Independent) Disks

There are several RAID levels numerated from 0 through 7.

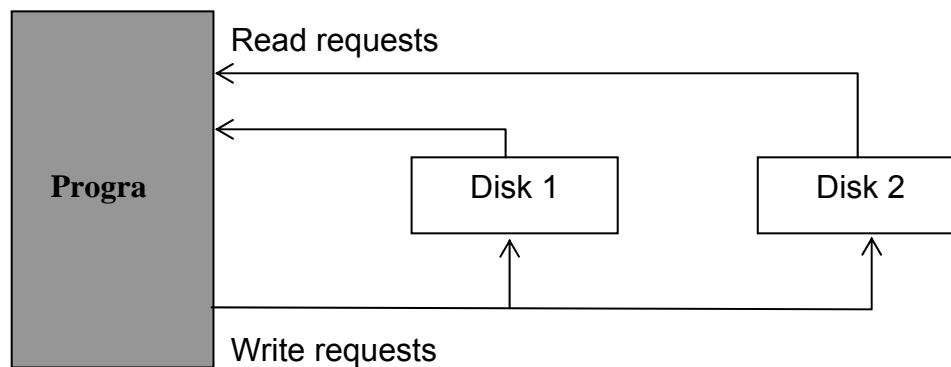
Level 0: *disk striping* - files are segmented; the segments are spread over several disks. This increases the performance of the system, if there is concurrent access to the same file: several users can access different segments of the same file because disk controller can transfer data independently.



RAID Level 0: Disk striping

Achieving High Reliability and High Performance (2)

Level 1: *disk mirroring* - all the data on a disk is simultaneously written to a second disk (or a second RAID set of disks). In the case of disk failure, the stored data can still be accessed. Write performance is equal to that of an individual drive. Read performance may double in the best case since read requests can be spread over two disks (or RAID sets) that can read independently.



RAID Level 1: Disk mirroring

This solution is quite expensive; only half of the total capacity is available for the data, the other half is consumed by redundant data. The implementation of RAID 1 is however simple.

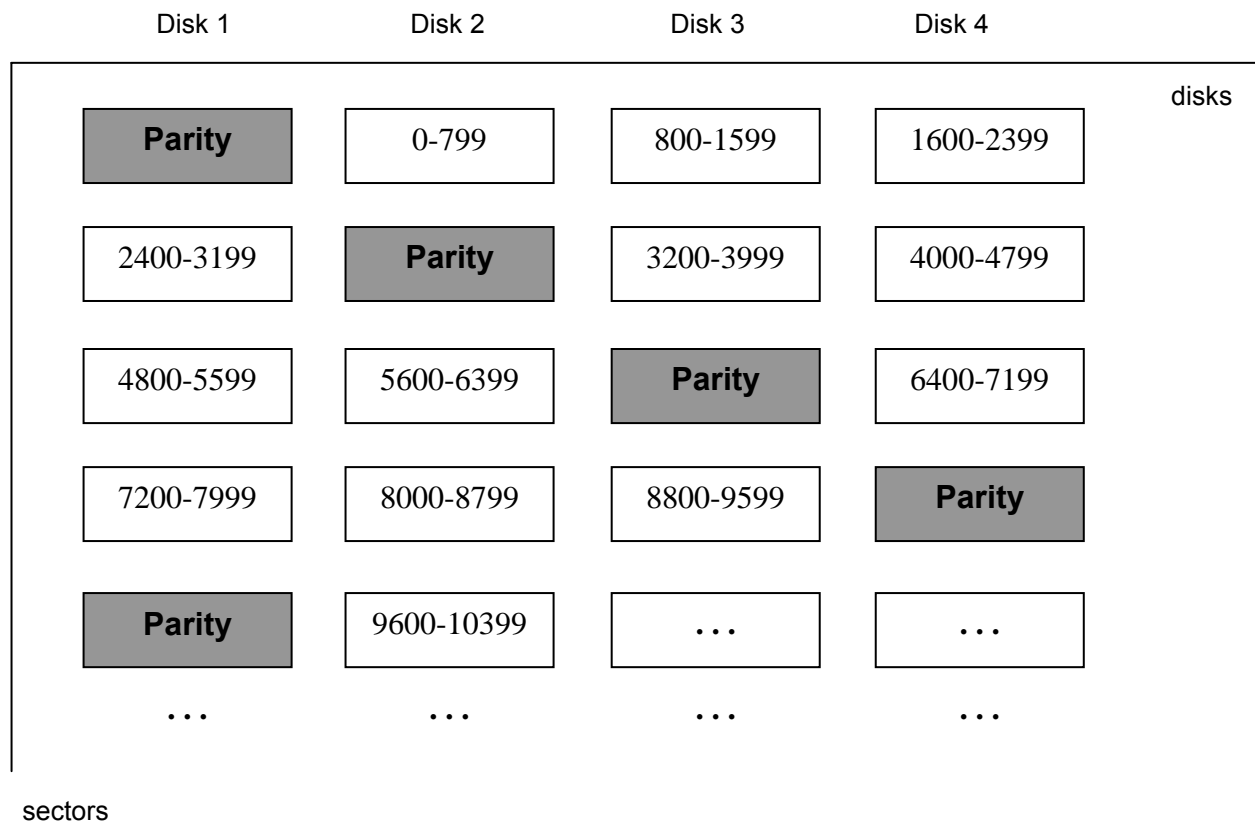
Mirroring of individual disks is available as software solution for various operating systems.

Achieving High Reliability and High Performance (3)

Levels 2-6: redundancy by introducing parity. In case of a disk failure, the lost data can be reconstructed from the data and parity information on the working disks. Cost is lower than in RAID 1.

The parity may be stored either on separate disks (RAID Levels 3,4,6 and 7), or on the same disks as normal data (Level 5).

Level 5: Parity is kept on all disks of the disk array. Let N denote the number of disks in the array. Then $(N-1)/N$ from the total capacity of the disks is used for useful data, the rest, $1/N$ of capacity, is intended for parity (redundant information).



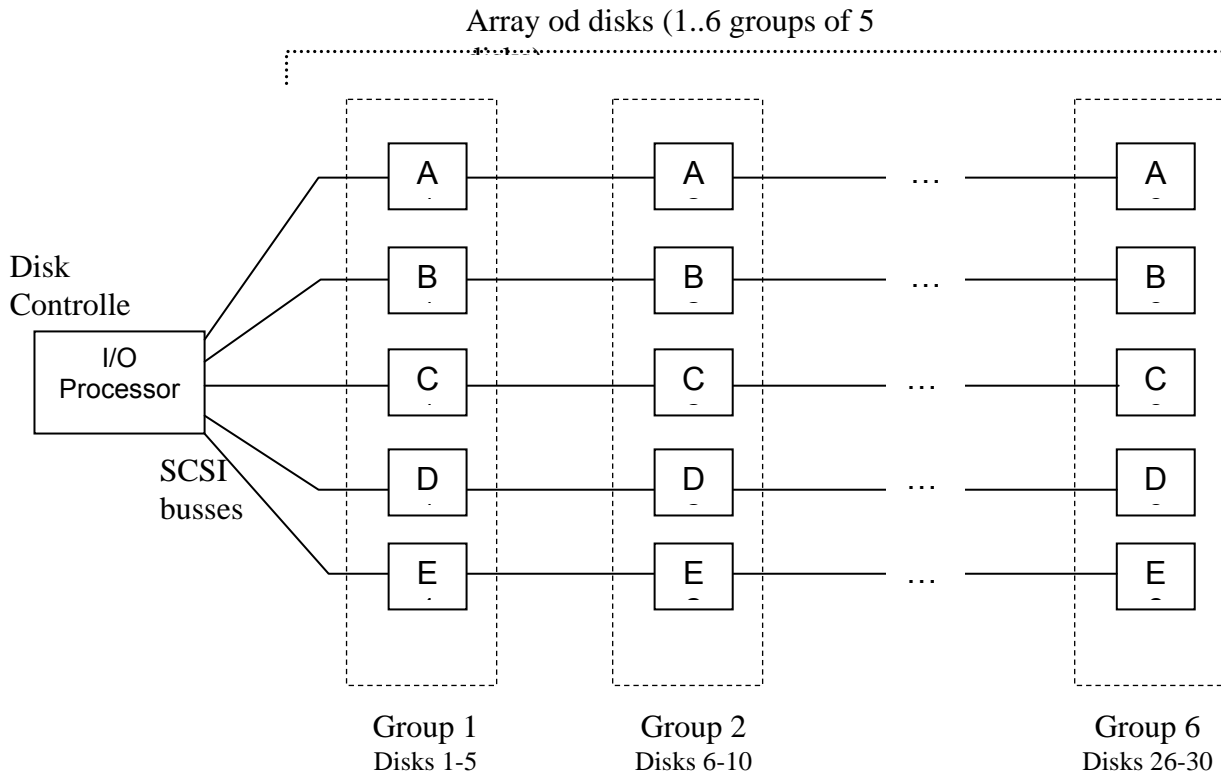
RAID Level 5

In RAID Levels 3,4,6 and 7 the parity drives can become a bottleneck of the system, as all write requests cause the parity information to be changed. In RAID Level 5, write requests can be processed faster, as there is no bottleneck at the parity drive, and read requests can also be processed faster, because information is spread over more drives.

Achieving High Reliability and High Performance (4)

HADA (High Availability Disk Array) - Data General

supports RAID 0, RAID 1 and RAID 5.



Conclusions:

- RAID offers a cost effective solution for configuring fault tolerant mass storage devices.
- RAID Level 5 is a good compromise between performance and reliability.
- RAID Levels 6 and 7 (independent bus and caches for each disk drive) offer highest performance, but are most expensive.