Chapter 16: File Structures and Hashing

CS-6360 Database Design

Dr. Chris Irwin Davis

Email: cid021000@utdallas.edu
Phone: (972) 883-3574
Office: ECSS 4.705

Outline

- **16.4** – Placing File Records on Disk
- **16.5** – Operations on Files
- **16.6** – Files of Unordered Records (Heap Files)
- **16.7** – File of Ordered Records (Sorted Files)
- **16.8** – Hashing techniques
- **16.9** – Other Primary File Organizations
- **16.10** – Parallelizing Disk Access Using RAID
- **16.11** – Modern Storage Architecture

- The following chapter 16 subsections are left to the student reader:
    - **16.1** – Introduction to Disks, Files, Hashing, and Storage
    - **16.2** – Secondary Storage Devices
    - **16.3** – Buffering of Blocks

16.4 Placing File Records on Disk

Files, Fixed-Length Records, and Variable-Length Records

- A file is a sequence of **records**.
- In many cases, all records in a file are of the *same record type*.
- If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records**.
- If different records in the file have different sizes, the file is said to be made up of **variable-length records**.

Placing File Records On Disk

- Data is usually stored in the form of **records**.
- Each record consists of a collection of related data **values** or **items**, where each value is formed of one or more bytes and corresponds to a particular **field** of the record.
- Records usually describe entities and their attributes.

Placing File Records On Disk

- For example, an EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as Name, Birth_date, Salary, or Supervisor.
- A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition.
- A **data type**, associated with each field, specifies the types of values a field can take.

Data Types and Size

- The data type of a field is usually one of the standard data types used in programming (e.g. numeric, string, Boolean, sometimes Date and/or Time).
- The number of bytes required for each data type is fixed for a given computer system.
    - Software platform dependent
    - Hardware platform dependent

Records

- For example, an EMPLOYEE record type may be defined —using the C programming language notation—as the following structure:

```
struct employee{
    char name[30];
    char ssn[9];
    int salary;
    int job_code;
    char department[20];
} ;
```

BLOBs

- The need may arise for storing data items that consist of large unstructured objects, which represent images, digitized video or audio streams, or free text.
- These are referred to as **BLOB**s (binary large objects).
- A BLOB data item is typically stored separately from its record in a pool of disk blocks, and a pointer to the BLOB is included in the record

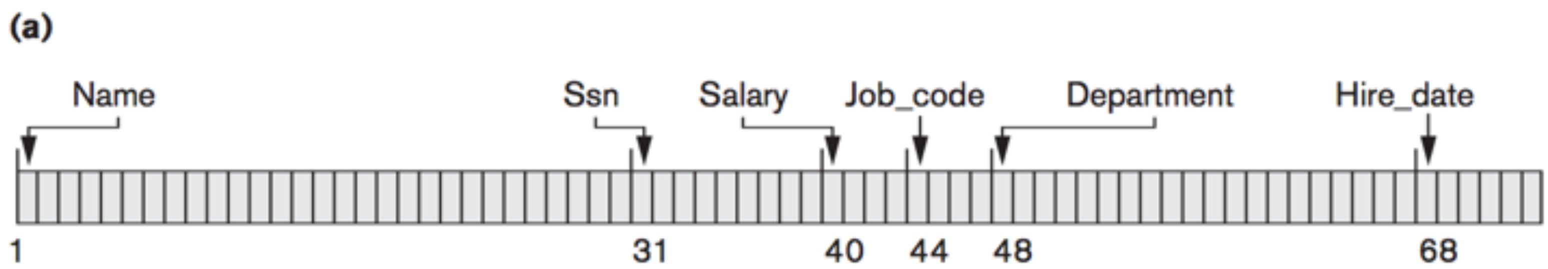
Variable-Length Records

- A file may have variable-length records for several reasons:
    - The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the Name field of EMPLOYEE can be a variable-length field.
    - The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a **repeating group**.

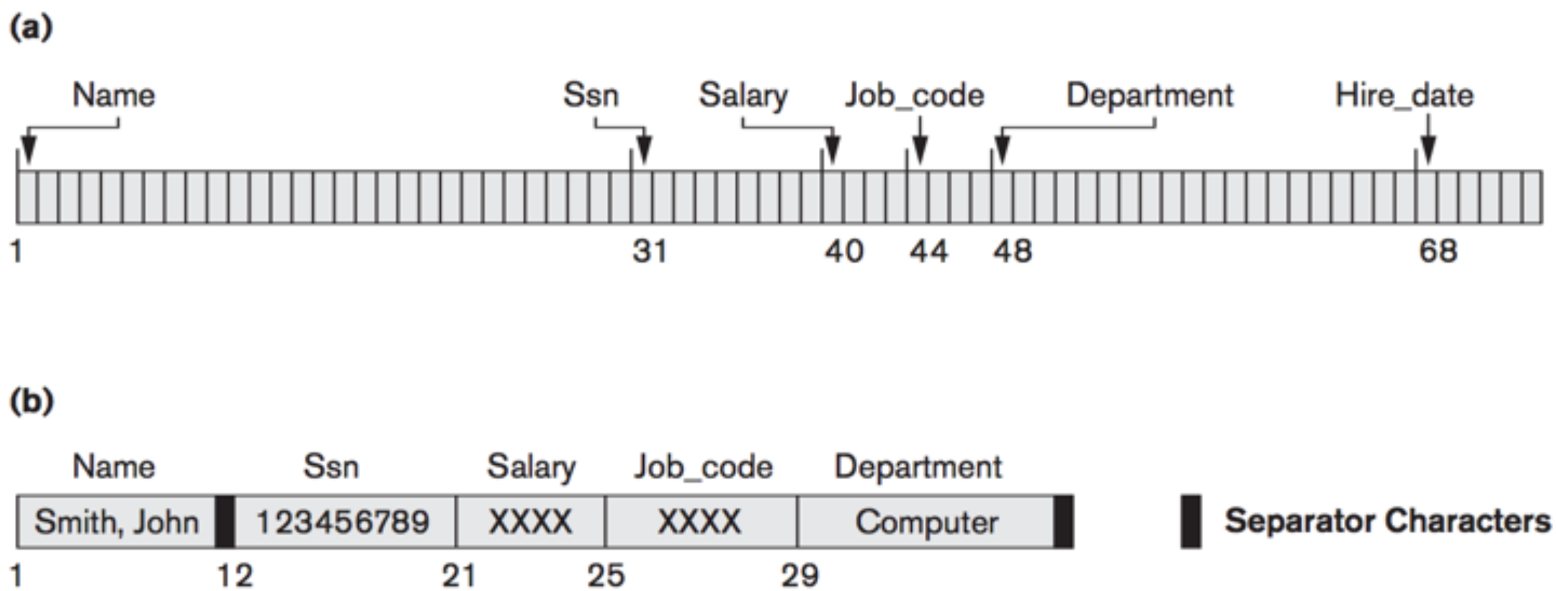
Variable-Length Records

- The file records are of the same record type, but one or more of the fields are **optional**; that is, they may have values for some but not all of the file records (**optional fields**).
- The file contains records of different record types and hence of varying size (**mixed file**). This would occur if related records of different types were clustered (placed together) on disk blocks; for example, the GRADE_REPORT records of a particular student may be placed following that STUDENT's record.

Record Storage Formats



Record Storage Formats



Record Storage Formats

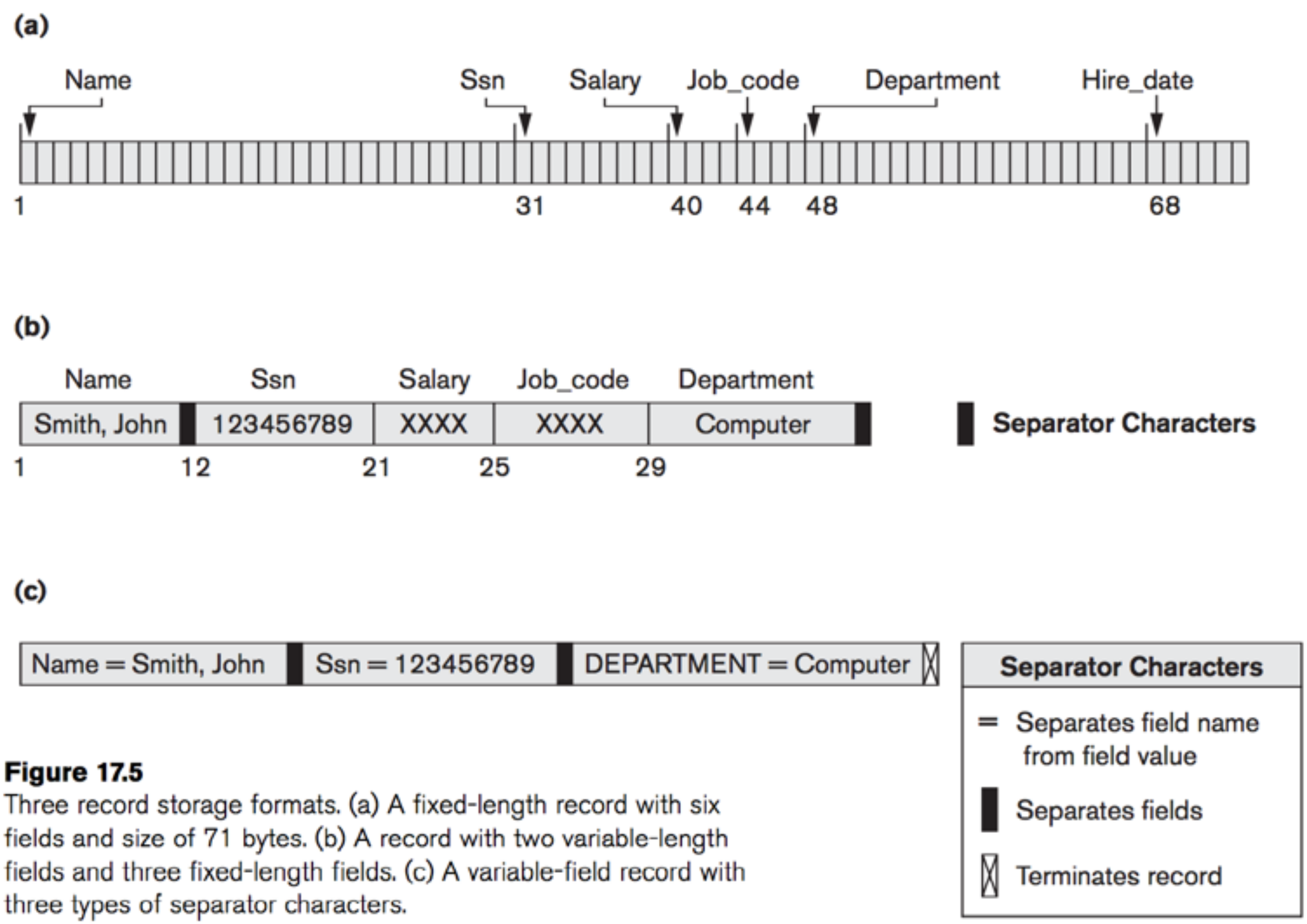


Figure 17.5
Three record storage formats. (a) A fixed-length record with six fields and size of 71 bytes. (b) A record with two variable-length fields and three fixed-length fields. (c) A variable-field record with three types of separator characters.

16.5 Operations on Files

Operations on Files

- **OPEN**: Readies the file for access, and associates a pointer that will refer to a current file record at each point in time.
- **FIND**: Searches for the first file record that satisfies a certain condition, and makes it the current file record.
- **FINDNEXT**: Searches for the next file record (from the current record) that satisfies a certain condition, and makes it the current file record.
- **READ**: Reads the current file record into a program variable.
- **INSERT**: Inserts a new record into the file & makes it the current file record.

Operations on Files

- **DELETE**: Removes the current file record from the file, usually by marking the record to indicate that it is no longer valid.
- **MODIFY**: Changes the values of some fields of the current file record.
- **CLOSE**: Terminates access to the file.
- **REORGANIZE**: Reorganizes the file records.
    - For example, the records marked deleted are physically removed from the file or a new organization of the file records is created.
- **FIND_ORDERED**: Retrieves all the records in the file in some specified order.
- **READ_ORDERED**: Read the file blocks in order of a specific field of the file.

16.6 Files of Unsorted Records (Heap Files)
16.7 Files of Sorted Records (Sorted Files)

Unordered Records (Heap Files)

- Also called a **heap** or a **pile** file.
- New records are inserted at the end of the file.
- A **linear search** through the file records is necessary to search for a record.
    - This requires reading and searching half the file blocks on the average, and is hence quite expensive.
- <u>Record insertion is quite efficient</u>
    - Just throw it on the heap
- <u>Reading</u> the records in order of a *particular field* requires sorting the file records.

Ordered Records (Sorted Files)

- Also called a **sequential** file.
- File records are kept sorted by the values of an *ordering field*.
- Insertion is expensive: records must be inserted in the correct order.
- It is common to keep a separate unordered *overflow* (or *transaction*) file for new records to improve insertion efficiency;
    - this is periodically merged with the main ordered file.

Ordered Records (Sorted Files)

- A **binary search** can be used to search for a record on its *ordering field* value.
- This requires reading and searching $\log_2$ of the file blocks on the average (i.e. binary search), an improvement over linear search.
- Reading the records in order of the ordering field is quite efficient.

Memory Blocks of a Sorted File

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
block 4	Allen, Troy					
	Anders, Keith					
	⋮					
	Anderson, Rob					
block 5	Anderson, Zach					
	Angeli, Joe					
	⋮					
	Archer, Sue					
block 6	Arnold, Mack					
	Arnold, Steven					
	⋮					
	Atkins, Timothy					
⋮						
block n –1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Manny					
block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Figure 17.7
Some blocks of an ordered (sequential) file of EMPLOYEE records with Name as the ordering key field.

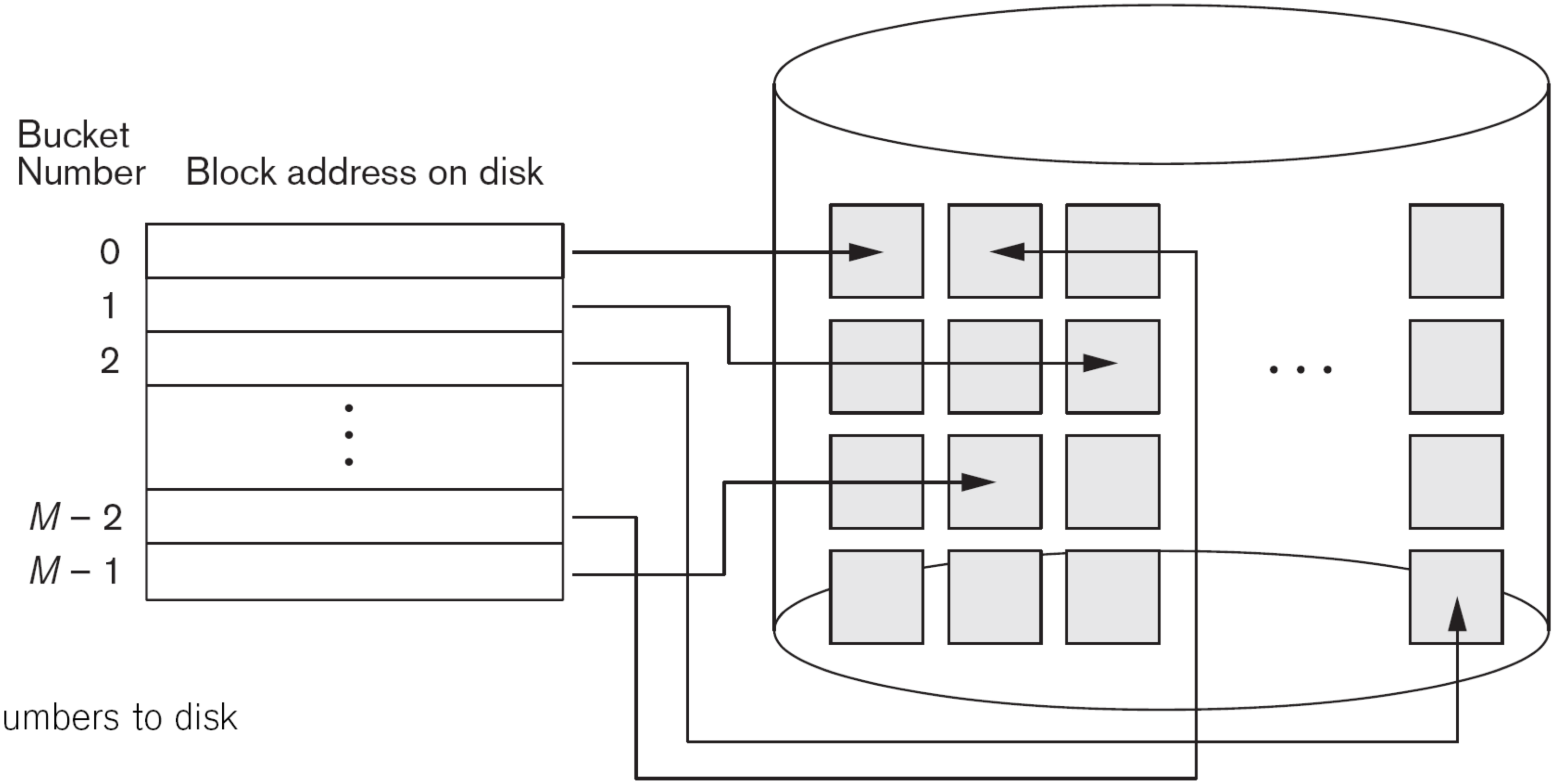


Figure 17.9
Matching bucket numbers to disk block addresses.

Other Strategies to Organize Data

- What if data were evenly distributed among memory “buckets”?
- Which field to sort / order by?
- How to insure even distribution?

16.8 Hashing

Hashing

- Another type of primary file organization is based on hashing, which provides very fast access to records under certain search conditions.

- This organization is usually called a **hash file**.

- The search condition must be an equality condition on a single field, called the **hash field**.

- In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**.

Hashing

- The idea behind hashing is to provide a function *h*, called a **hash function** or **randomizing function**, which is applied to the hash field value of a record and yields the address of the disk block in which the record is stored.

- A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Hashing

- Hashing is also used as an internal search structure within a program whenever a **group of records** is accessed exclusively by using the value of one field.
- We describe the use of hashing for
  - internal files in Section 16.8.1 – **Internal Hashing**;
  - then we show how it is modified to store external files on disk in Section 16.8.2 – **External Hashing**.
- In Section 16.8.3 we discuss techniques for extending hashing to dynamically growing files

Internal Hashing

- Hashing is typically implemented as a **hash table** through the use of an array of records.
- Suppose that the array index range is from 0 to $M - 1$, as shown in Figure 17.8(a); then we have M slots whose addresses correspond to the array indexes.
- We choose a hash function that transforms the hash field value into an integer between 0 and $M - 1$.

Common Hashing Function

- K is the Hash Field or Hash Key
- M is the number of available Blocks

$$h(K) = K \mathbf{\ mod\ } M$$

Alternate Hashing Functions

- **Folding**
  - applying an arithmetic function such as *addition*
  - a logical function such as *Exclusive OR*
  - different portions of the hash field value to calculate the hash address
  - Example
    - with an address space from 0 to 999 to store 1,000 keys, a 6-digit key 235469 may be folded and stored at the address:
    - (235+964) mod 1000 = 199)

Alternate Hashing Functions

- Pick some digits of the hash field value—for instance, the third, fifth, and eighth digits—to form the hash address
    - For example – storing 1,000 employees with Social Security numbers of 10 digits into a hash file with 1,000 positions would give the Social Security number 301-67-8923 a hash value of 172 by this hash function).
- Why not just hash on the first character?

What if a hash field is a non-integer?

Alternate Hashing Functions

- Non-integer hash field values can be transformed into integers before the mod function is applied.
- For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values.
- For a hash field whose data type is a string of 20 characters, the following algorithm can be used to calculate the hash address.

$temp \leftarrow 1;$
for $i \leftarrow 1$ to 20 do $temp \leftarrow temp * \text{code}(K[i\,])$ mod M ;
$hash\_address \leftarrow temp$ mod M;

Internal Hashing Data Structures

(a)

	Name	Ssn	Job	Salary
0				
1				
2				
3				
	⋮			
$M-2$				
$M-1$				

Figure 16.8

Internal hashing data structures. (a) Array of M positions for use in internal hashing.

What happens if a bucket gets Full?
Or target is already occupied?

Collision

- A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position, since its hash address is occupied.
- The process of finding another position is called **collision resolution**.
- There are numerous methods for collision resolution, including the following:

Hashed Files – Collisions

- There are numerous methods for collision resolution, including the following:
  - **Open addressing**: Proceeding from the occupied position specified by the hash address, the program checks the subsequent positions in order until an unused (empty) position is found.
  - **Chaining**: For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. In addition, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location.
  - **Multiple hashing**: The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

Other Hashing Issues

- The problem with most hashing functions is that they do not guarantee that distinct values will hash to distinct addresses, because…
  - the *hash field space*—the number of possible values a hash field can take—is usually much larger than the address space—the number of available addresses for records.
  - The hashing function maps the hash field space to the address space

Internal Hashing Data Structures

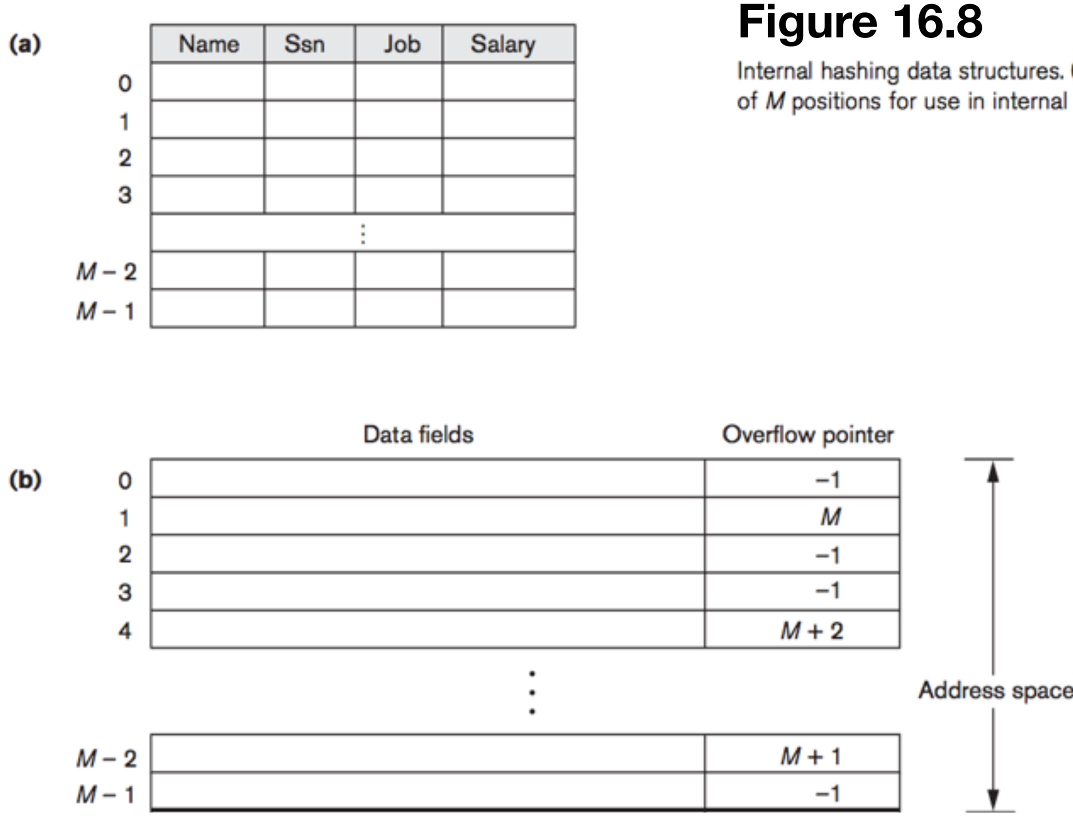


Figure 16.8
Internal hashing data structures. (a) Array of *M* positions for use in internal hashing.

Internal Hashing Data Structures

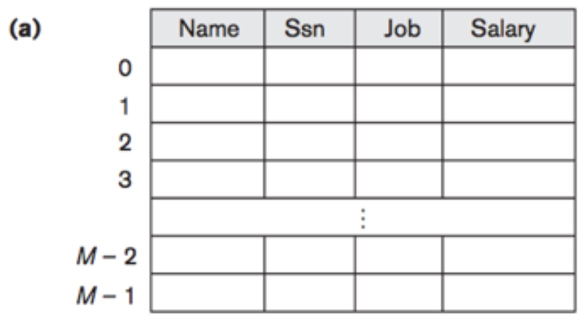
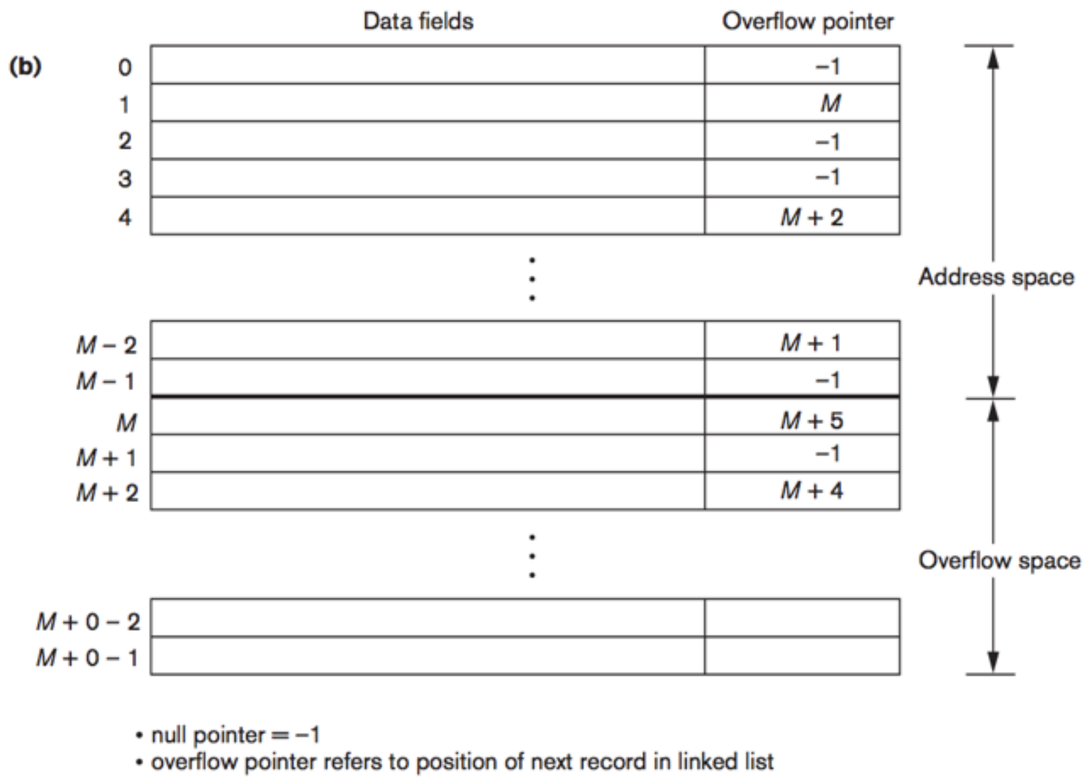


Figure 16.8

Internal hashing data structures. (a) Array of M positions for use in internal hashing. (b) Collision resolution by chaining records.



External Hashing (Hashed Files)

- Hashing for disk files is called **External Hashing**
- The file blocks are divided into *M* equal-sized **buckets**, numbered bucket_0, bucket_1, ..., bucket_{M-1}.
  - Typically, a bucket corresponds to one (or a fixed number of) disk blocks (*disk sectors*).
- One of the file fields is designated to be the **hash key** of the file.

External Hashing (Hashed Files)

- The record with hash key value K is stored in bucket i, where $i=h(K)$, and h is the **hashing function**.
- Search is very efficient on the hash key.
- Collisions occur when a new record hashes to a bucket that is already full.
  - An overflow file is kept for storing such records.

Hashed Files

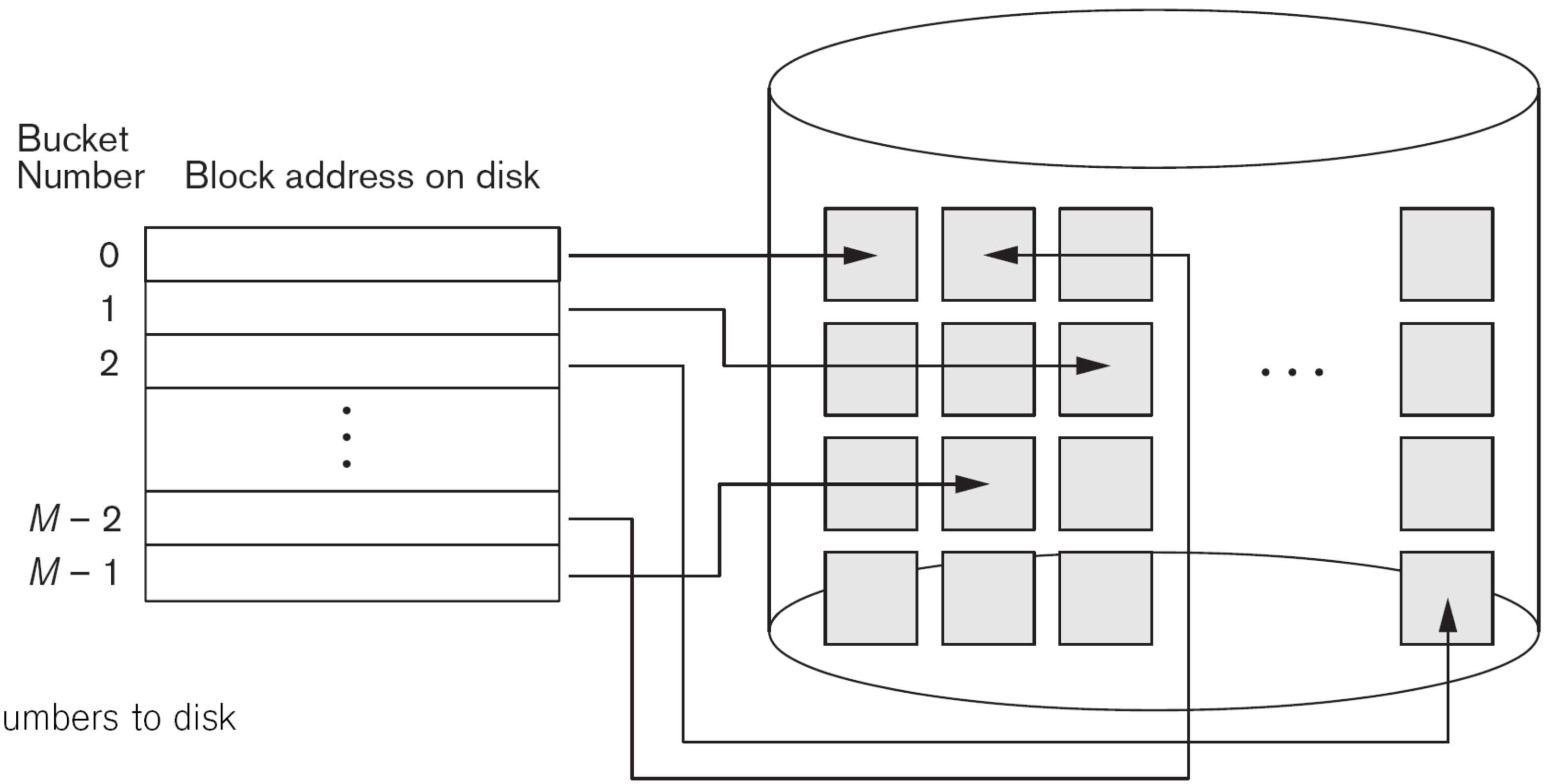


Figure 17.9
Matching bucket numbers to disk block addresses.

Hashed Files

- To reduce overflow records, a hash file is typically kept 70-80% full.
- The hash function *h* should distribute the records *uniformly* among the buckets
  - Otherwise, search time will be increased because many overflow records will exist.
- Main disadvantages of *static* external hashing:
  - *Fixed* number of buckets M is a problem if the number of records in the file grows or shrinks.
  - Ordered access on the hash key is inefficient (requires sorting the records).

Static Hashing

- The hashing scheme described so far is called *static* hashing because a *fixed* number of buckets M is allocated.
- This can be a serious drawback for dynamic files.
- Suppose that we allocate M buckets for the address space and let n be the maximum number of records that can fit in one bucket; then at most $(n * M)$ records will fit in the allocated space…

Static Hashing

- If the number of records turns out to be substantially fewer than (n * M), we are left with a lot of unused space.
- On the other hand, if the number of records increases to substantially more than (n * M), numerous collisions will result and retrieval will be slowed down because of the long lists of overflow records.
- In either case, we may have to change the number of blocks M allocated and then use a new hashing function (based on the new value of M) to redistribute the records.

Chaining – Common Bucket

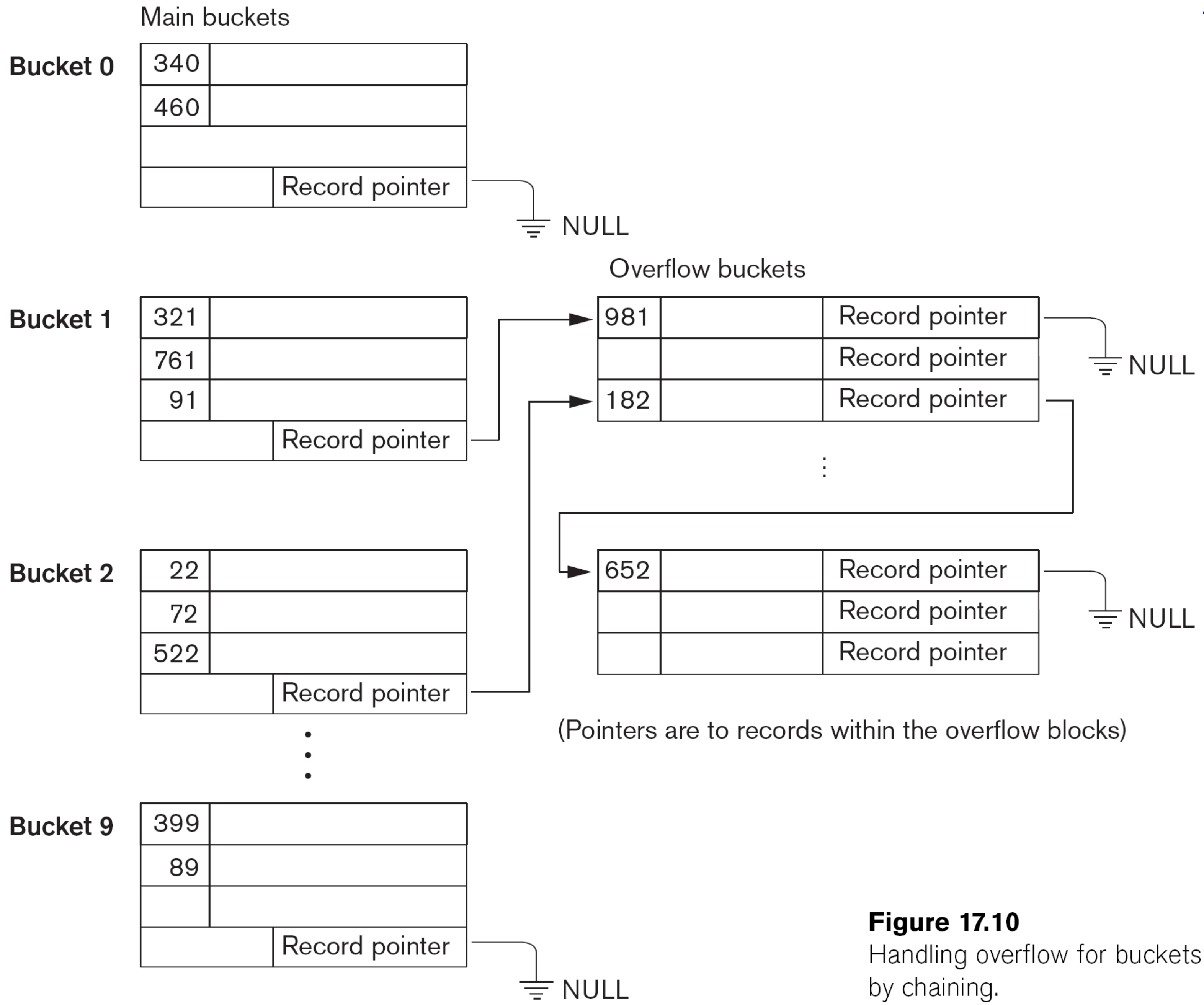


Figure 17.10
Handling overflow for buckets by chaining.

Hashing Techniques

- Dynamic and Extendible Hashing *Techniques*
  - Hashing techniques are adapted to allow the dynamic growth and shrinking of the *number* of file records.
  - These techniques include the following:
    - **Dynamic hashing**
    - **Extendible hashing**
    - **Linear hashing.**

Dynamic And Extendible Hashed Files

- Both dynamic and extendible hashing use the **binary representation** of the hash value $h(K)$ in order to access a **directory**.
  - In **dynamic hashing** the *directory is a binary tree*.
  - In **extendible hashing** the *directory is an array* of size 2^d where d is called the **global depth**.
  - The idea behind **linear hashing** is to allow a hash file to expand and shrink its number of buckets dynamically *without needing a directory*.

Dynamic Hashing

- These reorganizations can be quite time-consuming for large files.
- Newer **dynamic** file organizations based on hashing allow the number of buckets to *vary dynamically* with only localized reorganization

Dynamic Hashing

- Dynamic hashing, uses an access structure based on *binary tree* data structures.
- These hashing schemes take advantage of the fact that the result of applying a hashing function is a nonnegative integer and hence can be represented as a binary number.
- The access structure is built on the **binary representation** of the hashing function result, which is a string of **bits**. We call this the **hash value** of a record.
- Records are distributed among buckets based on the values of the *leading bits* in their hash values.

Dynamic Hashing

- The value of d can be increased or decreased by one at a time, thus doubling or halving the number of entries in the directory array.
  - **Doubling** is needed if a bucket, whose local depth d' is equal to the global depth d, overflows.
  - **Halving** occurs if $d > d'$ for all the buckets after some deletions occur.
- Most record retrievals require two block accesses—one to the directory and the other to the bucket.

Dynamic Hashing

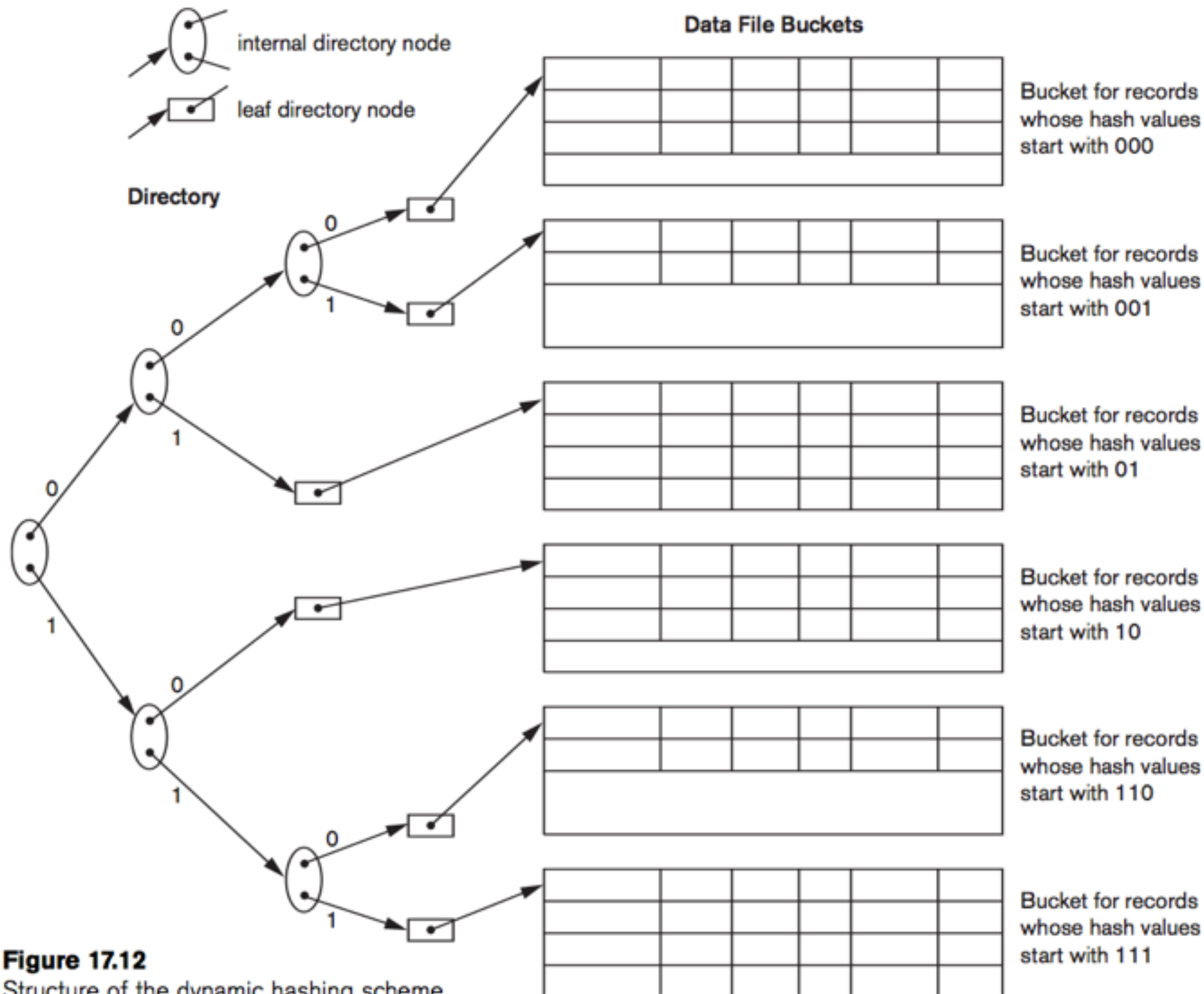


Figure 17.12
Structure of the dynamic hashing scheme.

Extendible Hashing

- There does not have to be a distinct bucket for each of the 2^d directory locations.
- Several directory locations with the same first d' bits for their hash values may contain the same bucket address if all the records that hash to these locations fit in a single bucket.
- A local depth d'—stored with each bucket—specifies the number of bits on which the bucket contents are based.
- The slide after next shows a directory with global depth $d = 3$.

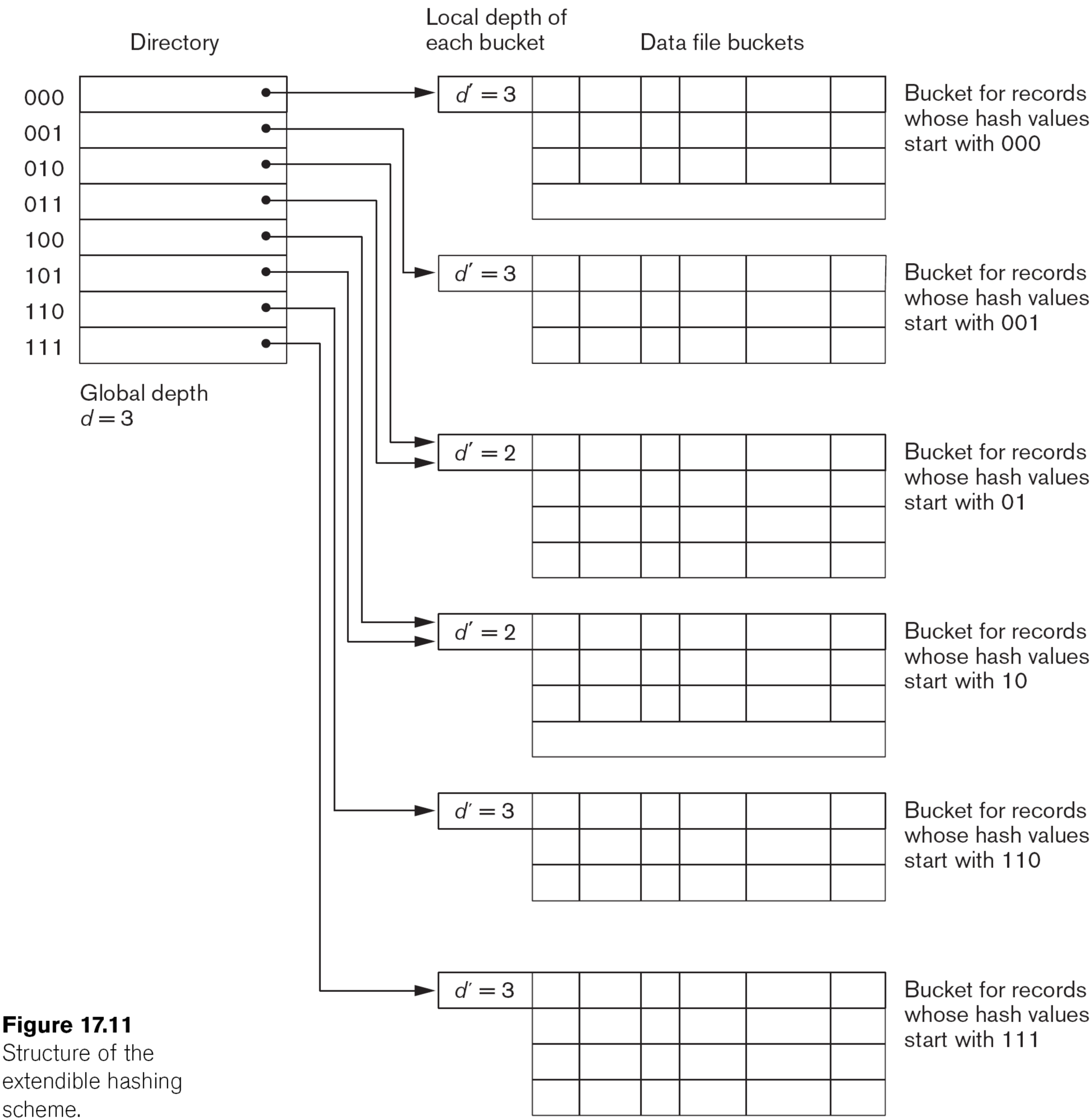


Figure 17.11
Structure of the extendible hashing scheme.

16.10 RAID

Parallelizing Disk Access using RAID Technology

- Secondary storage technology must take steps to keep up in performance and reliability with processor technology.
- A major advance in secondary storage technology is represented by the development of **RAID**, which originally stood for **Redundant Arrays of Inexpensive Disks**.
- The main goal of RAID is to even out the widely different rates of performance improvement of disks against those in memory and microprocessors.

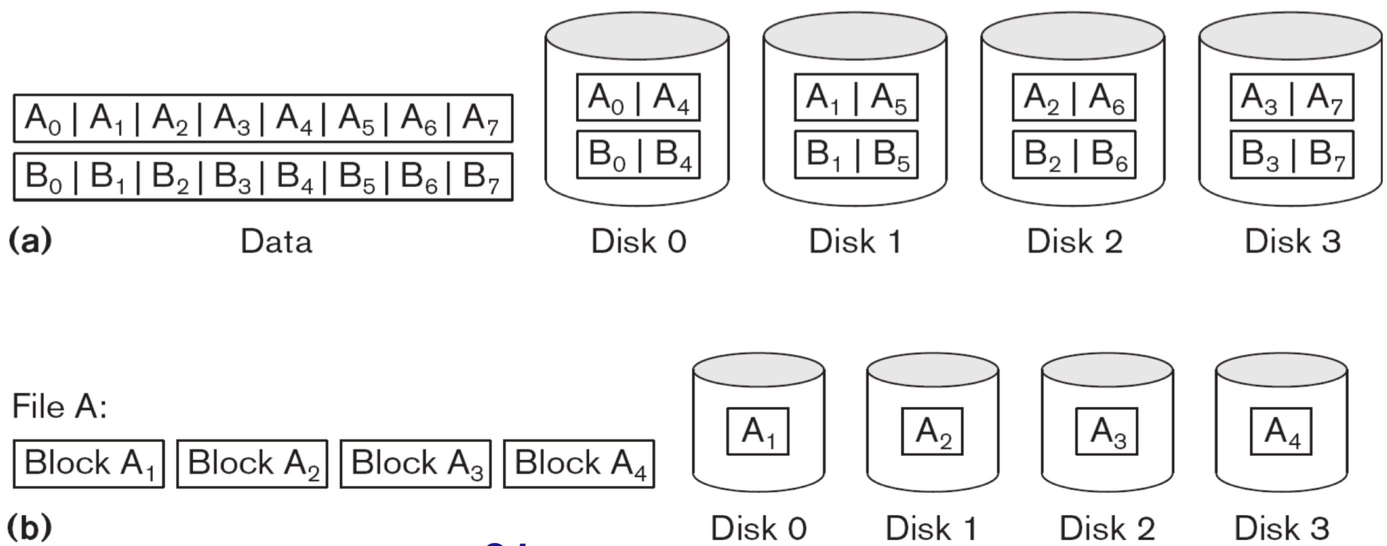
Three High-Level RAID Strategies

- **Redundancy**
  - Reliability
- **Data striping**
  - Performance
- **Parity**
  - Error detection (vs. error correction)

RAID Technology (cont.)

- A natural solution is a large array of small independent disks acting as a single higher-performance logical disk.
- A concept called **data striping** is used, which utilizes parallelism to improve disk performance.
- Data striping distributes data transparently over multiple disks to make them appear as a single large, fast disk.

Figure 17.13
Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.



RAID Technology (cont.)

- Different RAID organizations/configurations were defined based on different combinations of the two factors of granularity of data interleaving (striping) and pattern used to compute redundant information.
  - **Raid level 0** has no redundant data and hence has the best write performance at the risk of data loss
  - **Raid level 1** uses mirrored disks.
  - **Raid level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components. Level 2 includes both error detection and correction.

RAID Technology (cont.)

- **Raid level 3** uses a single parity disk relying on the disk controller to figure out which disk has failed.
- **Raid Levels 4 and 5** use block-level data striping, with level 5 distributing data and parity information across all disks.
- **Raid level 6** applies the so-called P + Q redundancy scheme using Reed-Soloman codes to protect against up to two disk failures by using just two redundant disks.

Use of RAID Technology (cont.)

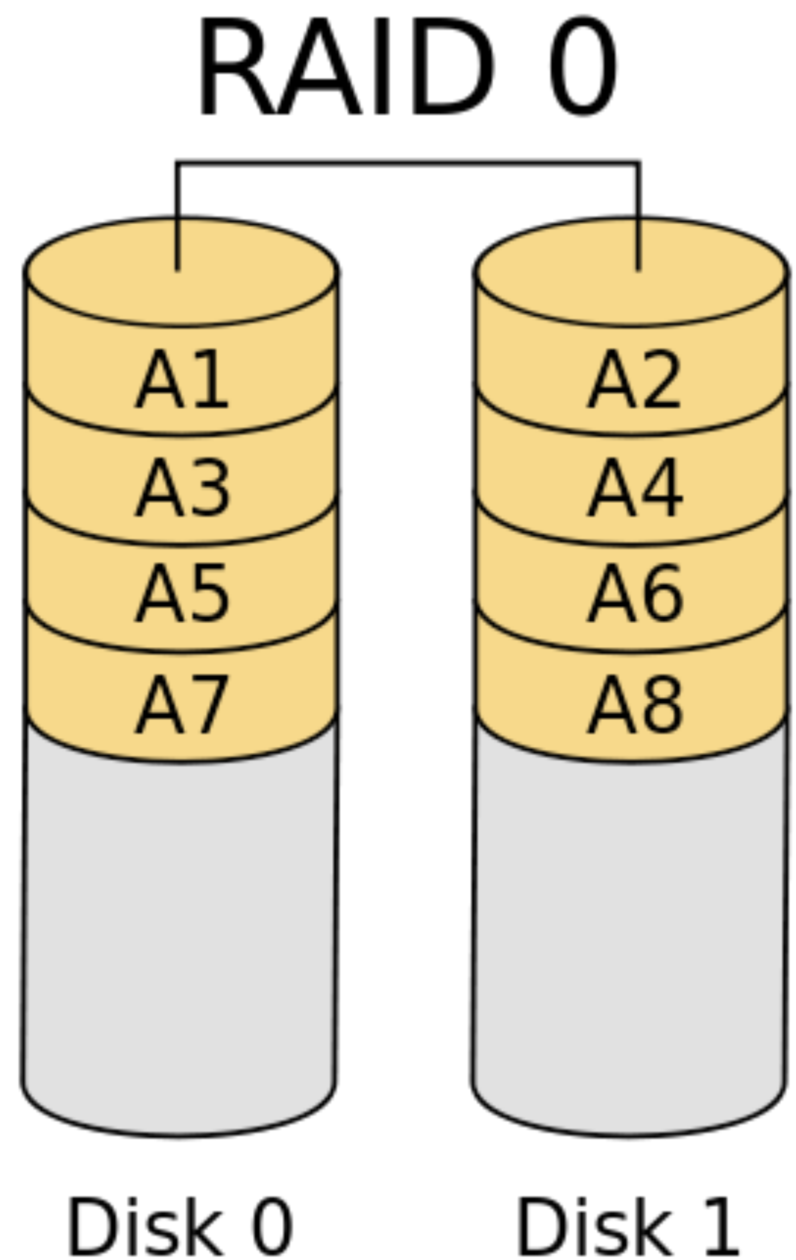
- Different RAID organizations are being used under different situations
  - **RAID level 1** (mirrored disks) is the easiest for rebuild of a disk from other disks
    - It is used for critical archiving applications like logs
  - **RAID level 2** uses memory-style redundancy by using Hamming codes, which contain parity bits for distinct overlapping subsets of components.
    - Level 2 includes both error detection and correction.
  - **RAID level 3** (single parity disks relying on the disk controller to figure out which disk has failed)
    - *and*
  - **RAID level 5** (block-level data striping)
    - are preferred for Large volume storage, with level 3 giving higher transfer rates.

Use of RAID Technology (cont.)

- Most popular uses of the RAID technology currently are:
  - **Level 0** (with striping), **Level 1** (with mirroring) and **Level 5** with an extra drive for parity.
- Design Decisions for RAID include:
  - Level of RAID
  - Number of disks
  - Choice of parity schemes
  - Grouping of disks for block-level striping

RAID 0

- **RAID 0** splits data evenly across two or more disks (striped) without parity information for speed.
- **RAID 0** was not one of the original RAID levels and provides no data redundancy.
- **RAID 0** is normally used to increase performance, although it can also be used as a way to create a large logical disk out of two or more physical ones.

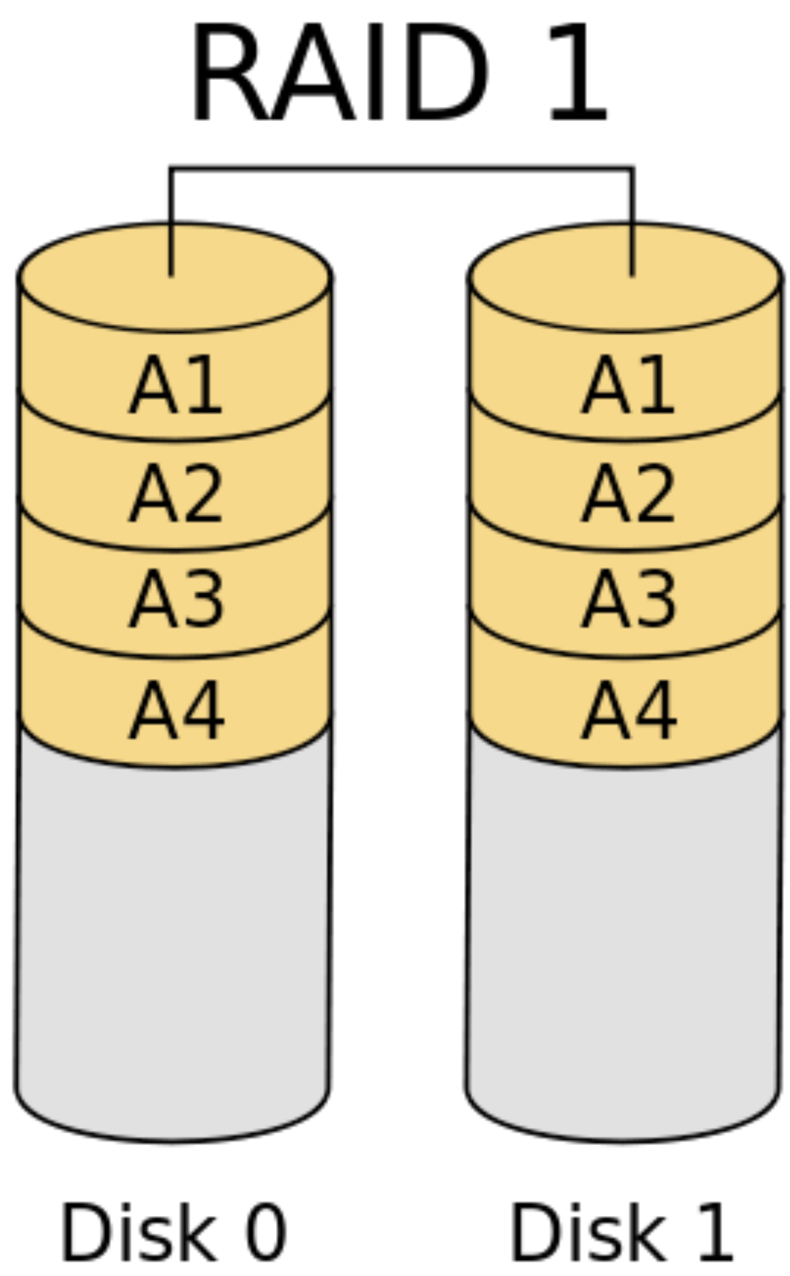


RAID 0 Performance

- RAID 0 is also used in some computer gaming systems where performance is desired and data integrity is not very important.
- However, real-world tests with computer games have shown that RAID-0 performance gains are minimal, although some desktop applications will benefit.
- Striping does not *always* increase performance (in certain situations it will actually be slower than a non-RAID setup), but in most situations it will yield a significant improvement in performance.

RAID 1

- An exact copy (or mirror) of a set of data on two disks.
- This is useful when read performance or reliability is more important than data storage capacity.
- Such an array can only be as big as the smallest member disk.
- A classic RAID 1 mirrored pair contains two disks over a single disk.



RAID 1 Performance

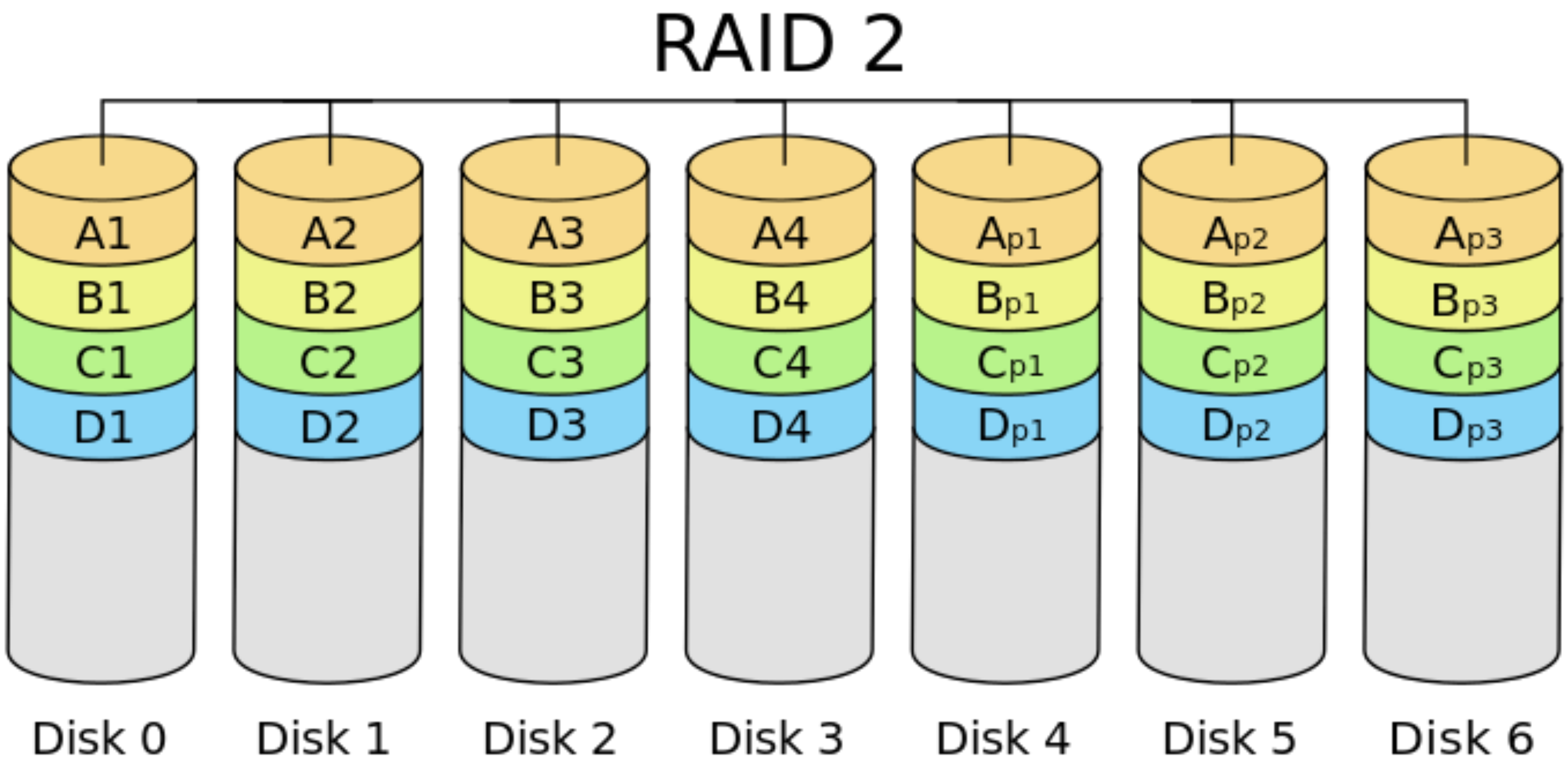
- Since each member contains a complete copy and can be addressed independently, ordinary wear-and-tear reliability is raised by the power of the number of self-contained copies.
- Since all the data exists in two or more copies, each with its own hardware, the read performance can go up roughly as a linear multiple of the number of copies.
- To maximize performance benefits of RAID 1, independent disk controllers are recommended, one for each disk.

Rare and Deprecated RAID Levels
RAID 2, 3, & 4

RAID 2 (deprecated)

- RAID 2 **bit-level striping** (rather than block) level, and uses a Hamming code for *error correction*.
- The disks are synchronized by the controller to spin at the same angular orientation (they reach Index at the same time), so it generally cannot service multiple requests simultaneously.
- Extremely high data transfer rates are possible.
- ***This is the only original level of RAID that is not currently used.***

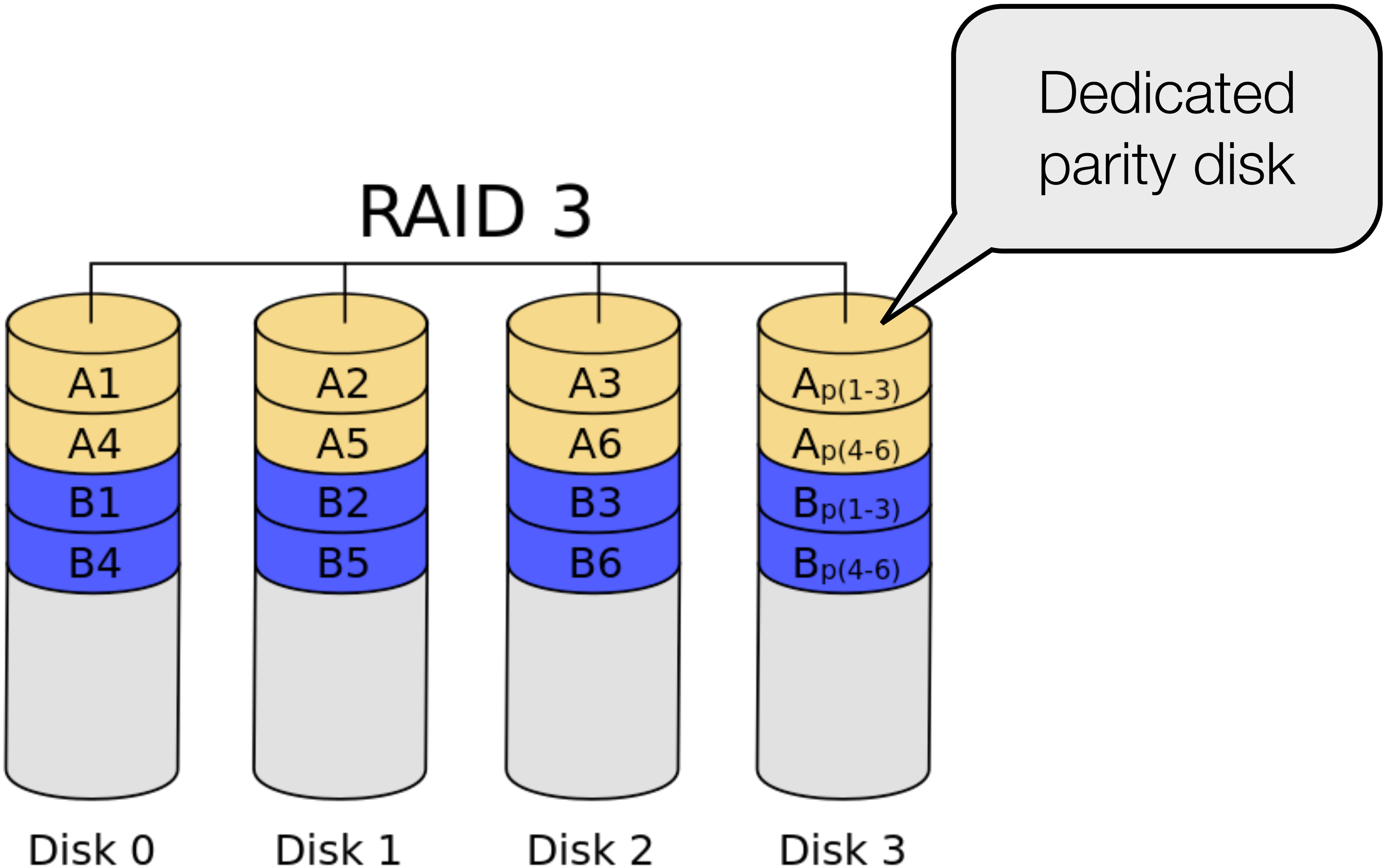
RAID 2


RAID 2
A1
A2
A3
A4
Ap1
Ap2
Ap3
B1
B2
B3
B4
Bp1
Bp2
Bp3
C1
C2
C3
C4
Cp1
Cp2
Cp3
D1
D2
D3
D4
Dp1
Dp2
Dp3
Disk 0
Disk 1
Disk 2
Disk 3
Disk 4
Disk 5
Disk 6

RAID 3 (mostly deprecated)

- **RAID 3 byte-level striping** with a dedicated parity disk.
- RAID 3 is very rare in practice.
- One of the characteristics of RAID 3 is that it generally cannot service multiple requests simultaneously.
    - This happens because any single **block** of data will, by definition, be spread across all members of the set and will reside in the same location.

RAID 3 (mostly deprecated)

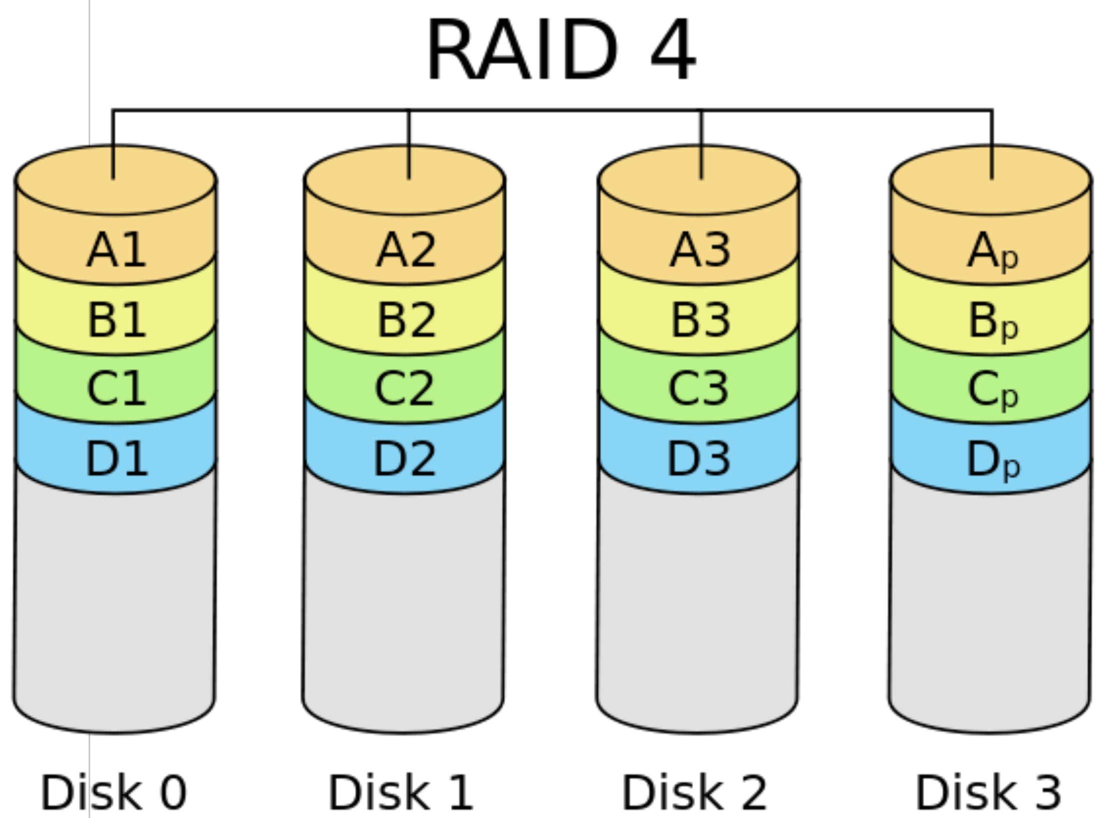


RAID 4

- **RAID 4** uses *block-level striping* with a dedicated parity disk.
- This allows each member of the set to act independently when only a single block is requested.
- Very uncommon, but one enterprise level company that has previously used it is NetApp.

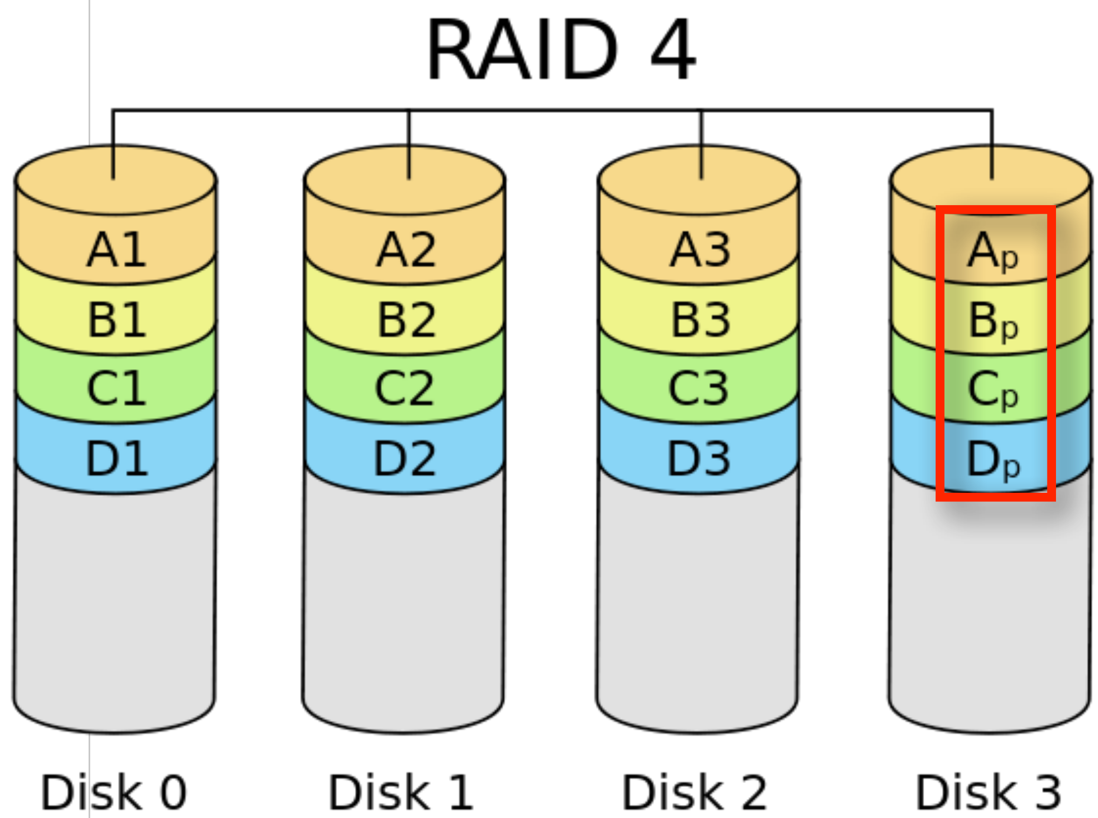
RAID 4

- In the example on the right, a read request for block A1 would be serviced by disk 0.
- A simultaneous read request for block B1 would have to wait, but a read request for B2 could be serviced concurrently by disk 1.



RAID 4

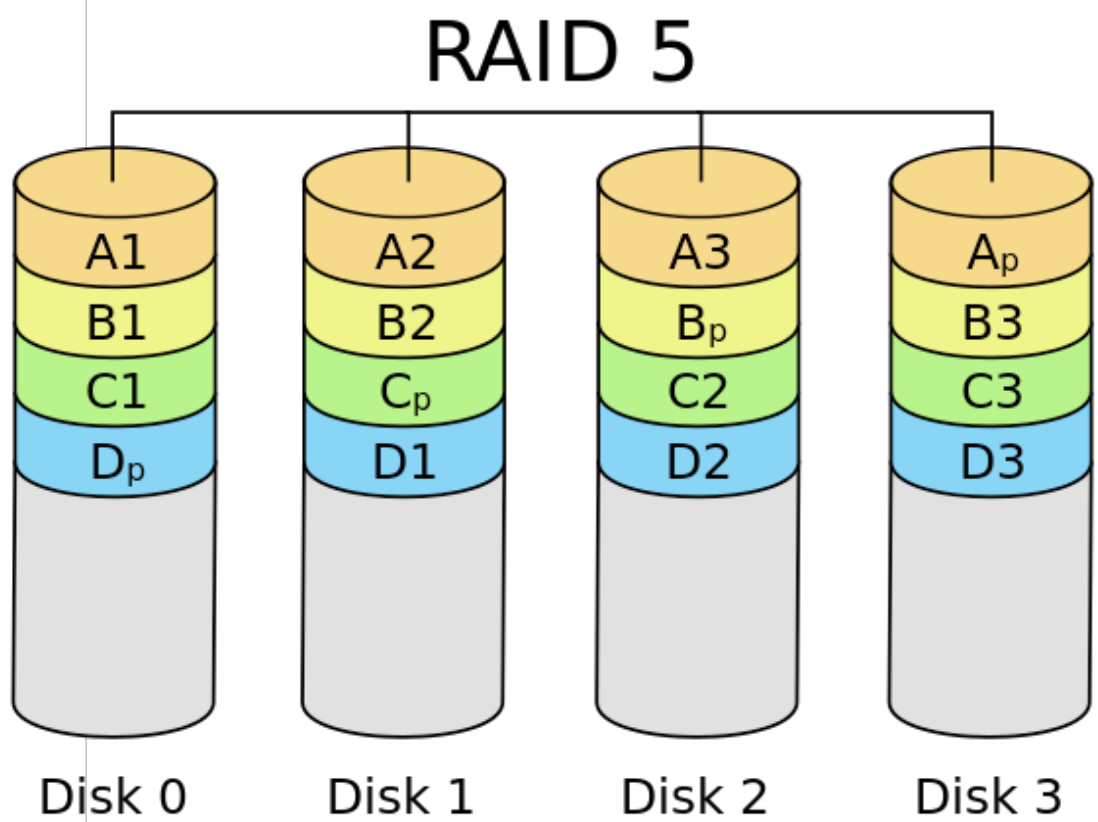
- In the example on the right, a read request for block A1 would be serviced by disk 0.
- A simultaneous read request for block B1 would have to wait, but a read request for B2 could be serviced concurrently by disk 1.



RAID 5 & 6

RAID 5

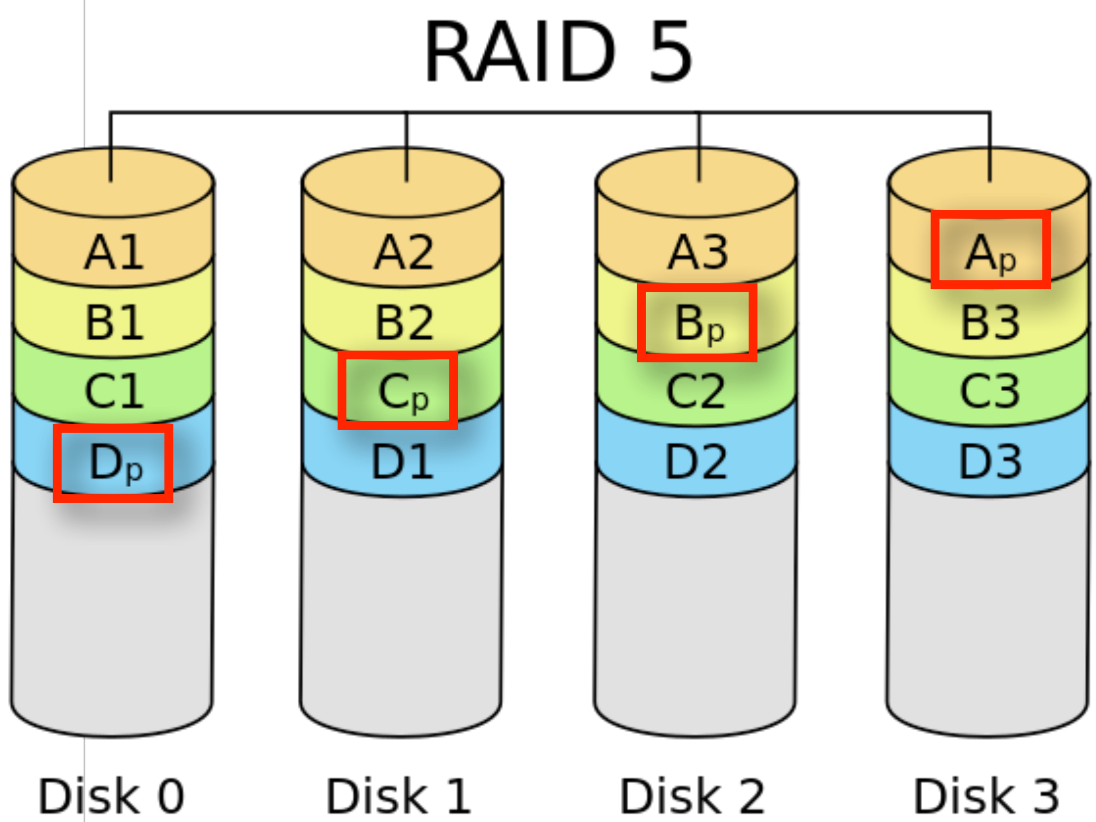
- **RAID 5** uses block-level striping with parity data distributed across all member disks
- Diagram of a **RAID 5** setup with distributed parity with each color representing the group of blocks in the respective parity block (a stripe).



RAID 5

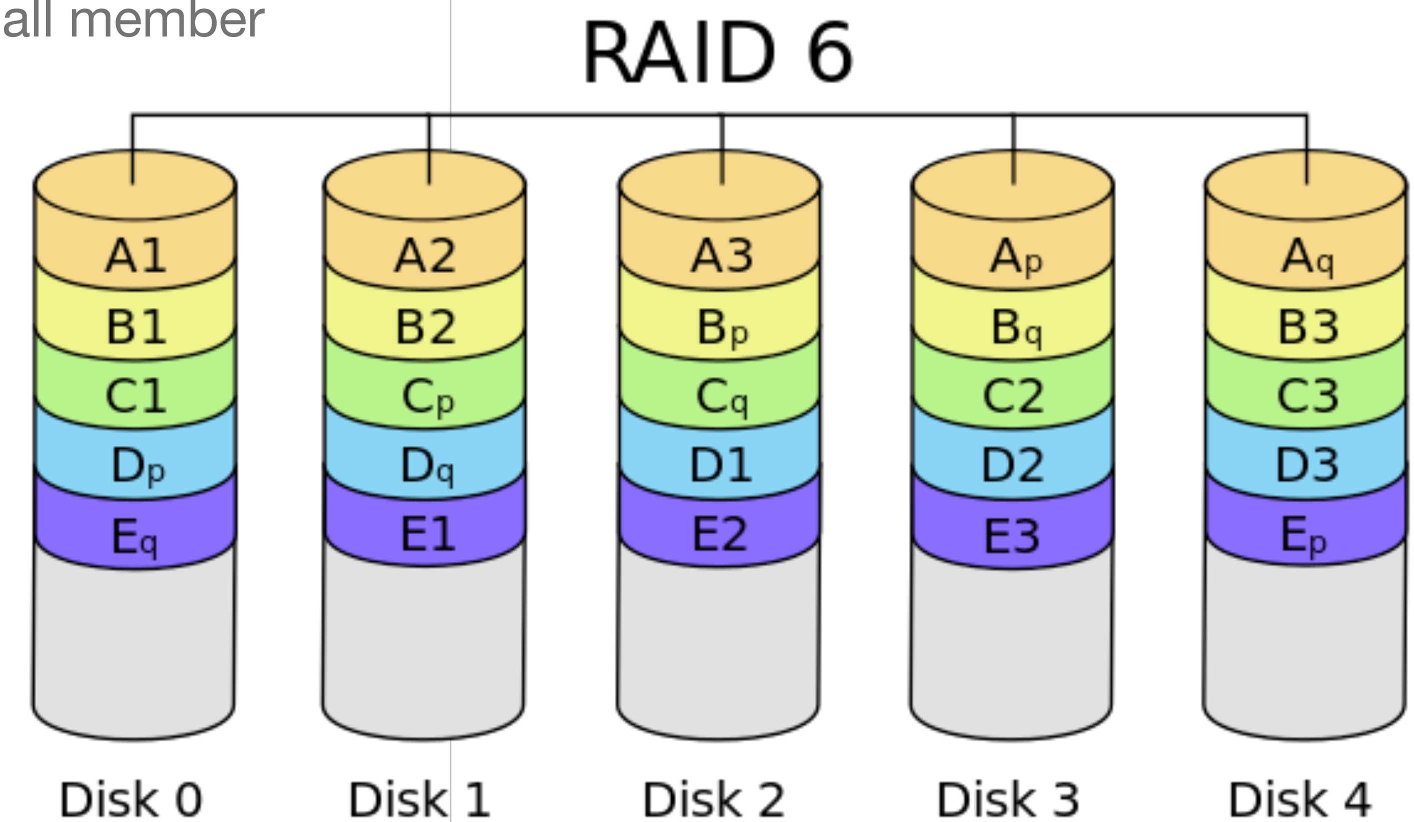
- **RAID 5** uses block-level striping with parity data distributed across all member disks

- Diagram of a **RAID 5** setup with distributed parity with each color representing the group of blocks in the respective parity block (a stripe).



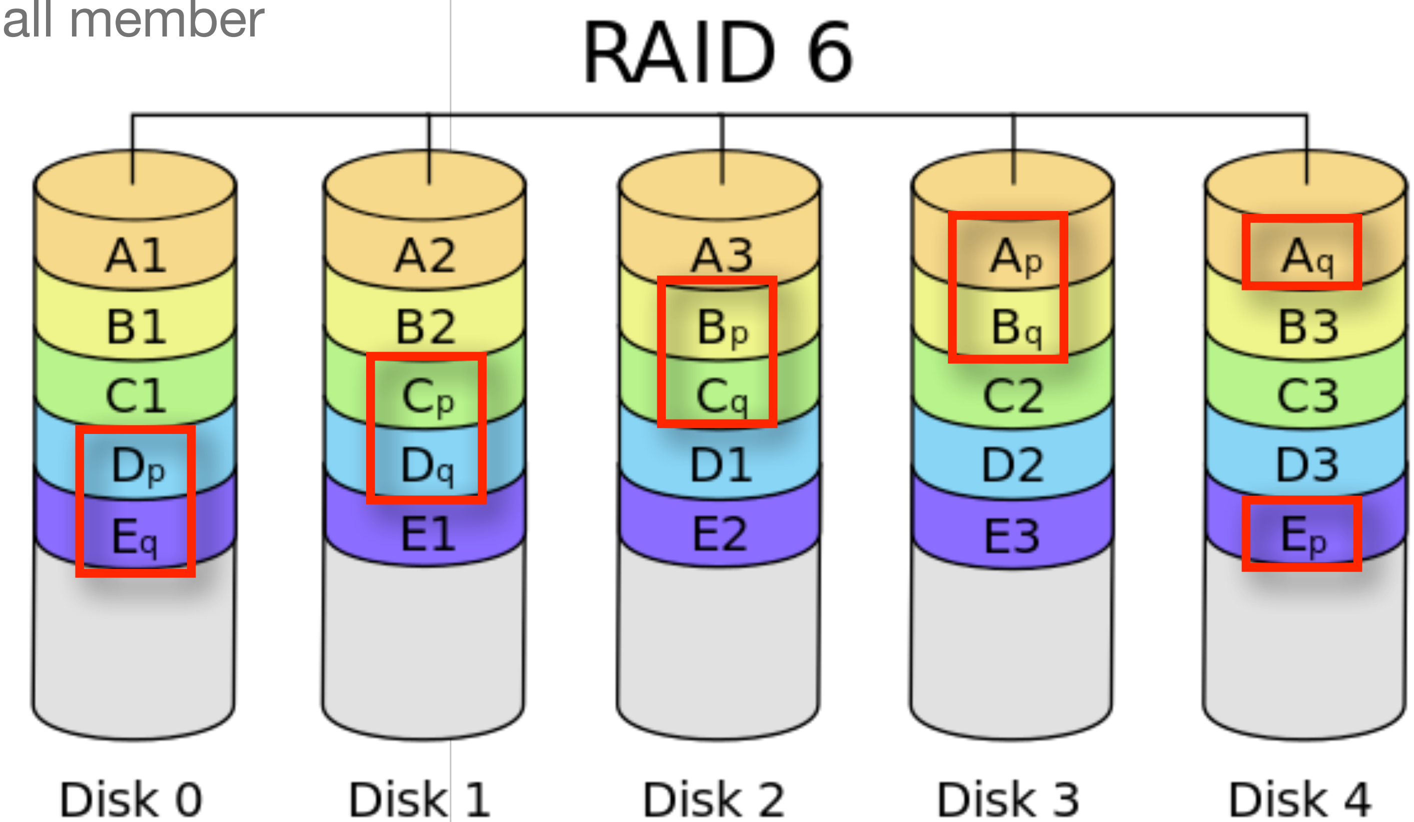
RAID 6

- **RAID 6** extends RAID 5 by adding an additional parity block; thus it uses block-level striping with *two* parity blocks distributed across all member disks.



RAID 6

- **RAID 6** extends RAID 5 by adding an additional parity block; thus it uses block-level striping with *two* parity blocks distributed across all member disks.



How Does Parity Work?

Parity Recovery

- Exclusive OR (XOR)

A	B	A $\oplus$ B
0	0	0
0	1	1
1	0	1
1	1	0

Parity Recovery

- Associativity
  - $A \oplus B \oplus C = (A \oplus B) \oplus C$
  - $A \oplus B \oplus C = A \oplus (B \oplus C)$
- “True” if cardinality of True operands is odd
- Scalable to any number of operands (i.e. disk bits)

Parity Recovery

A	B	C	A $\oplus$ B $\oplus$ C
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Parity Recovery

A	B	C	$A \oplus B \oplus C$
1	0	1	?
1	1	?	0
1	?	0	0
?	0	1	1

Parity Recovery

- Example with nybbles

A	B	C	$A \oplus B \oplus C$
1100	1011	0110	0001
0111	1001	1100	0010
1101	0101	1000	0000
0011	0100	1000	1111

Parity Recovery

- Example with nybbles

A	B	C	A ⊕ B ⊕ C
1101	0101	1000	????
1011	1100	????	0001
0111	????	1100	0010
????	1111	1000	0011

Parity Recovery

- Example with nybbles

A	B	C	$A \oplus B \oplus C$
1101	0101	1000	0000
1011	1100	0110	0001
0111	1001	1100	0010
0100	1111	1000	0011