Today:

- RAID and error recovery for disks.

Today and Next:

- Representing important elements of databases: attributes, tuples, relations, objects, etc.

Soon:

Data structures that allow efficient queries and modifications of data:

- Indexed-sequential files, other index structures.
- B-trees.
- Hash tables.
- Multidimensional index structures.

Redundant Arrays of Independent Disks

Disks fail with a mean time to failure of about 10 years.

- Typically, failure rate is greatest early and late.
- Redundancy allows data to survive a disk crash.
- Techniques identified by "RAID levels."
- RAID level 1 = mirror the disk.

MTTF for Mirrored Disks

Assume:

- 5% failure per year; MTTF = 10 years.
- 3 hours to replace and restore failed disk.

If a failure to one disk occurs, then the other better not fail in the next three hours.

- Probability of failure = $5\% \times 3/(24 \times 365) = 1/58400$.
- If one disk fails every ten years, then one of two will fail every 5 years.
- One in 58,400 of those failures results in data loss; MTTF = 292,000 years.

Modulo-2 Arithmetic and Parity Checks

More powerful forms of RAID involve parity checks. Central idea is modulo-2 addition of bit vectors.

- $x \oplus y$ = bitwise modulo-2 sum of corresponding positions of bit vectors x and y.
- **Example**: $11110000 \oplus 10101010 = 01011010$.

Properties of $\oplus$

- *Commutativity*: $x \oplus y = y \oplus x$.
- *Associativity*: $x \oplus (y \oplus z) = (x \oplus y) \oplus z$.
- *Self-inverse*: $x \oplus x = \bar{0}$ ($\bar{0}$ = all-0 vector).
- *Identity*: $x \oplus \bar{0} = \bar{0} \oplus x = x$.

Useful Consequences

Suppose $\bar{0} = x_1 \oplus x_2 \oplus \cdots \oplus x_n$. Then ($\oplus$-ing x_i to each side):

- $x_i = x_1 \oplus \cdots \oplus x_{i-1} \oplus x_{i+1} \oplus \cdots \oplus x_n$.
  - We call any of the x_i's the *parity check* for the others.
  - The modulo-2 sum of some vectors and their parity check is always 0.
  - Put another way, the vectors and their parity check have an even number of 1's in each position.

RAID Level 4

Use n *data disks* and one *redundant disk*.

- Each block of the redundant disk is the parity check for the corresponding blocks of the data disks.

Example

$n = 3$, blocks are 8 bits.

- First blocks of data disks are: 11110000, 10101010, 00111000.

- Thus, the first block of the redundant disk is $11110000 \oplus 10101010 \oplus 00111000 = 01100010$.

Recovery From Error

We must be able to restore whatever disk crashes.

- Just compute the modulo-2 sum of corresponding blocks of the other disks.

Example

Disk 2 crashes. First blocks are:
11110000, ????????, 00111000, 01100010.

Fill in ?'s by $\oplus$ applied to other three:
???????? = 10101010.

Reading

Read data disk normally.

- Interesting possibility: If we want to read from disk i, but it is busy and all other disks are free, then instead read corresponding blocks from all other disks and modulo-2 sum them.

Writing

To write block j of data disk i (new value $= x$):

1. Read old value of that block, say y.
2. Read the jth block of the redundant disk, say z.
3. Compute $w = x \oplus y \oplus z$.
4. Write x in block j of disk i.
5. Write w in block j of the redundant disk.
   - ❖ Total of four disk I/O's required.

Problem

Why does this work?

- Intuition: $x \oplus y$ is the "change" to the data. Redundant disk must change to compensate.

RAID Level 5

- Problem: on writes, redundant disk is a bottleneck.
- Solution: vary the redundant disk for different blocks.
  - Example: n disks; block j is redundant on disk i if i = remainder of j/n.

Error Correcting Codes

RAID level 6 requires the use of error-correcting codes to handle more than one simultaneous disk failure.

- *Code* = set of "legal" bit vectors (*code words*).
  - ✥ **Example**: requiring a parity check says that the code words are those with an even number of 1's, if we regard a code word as corresponding bits from each disk.
- *Hamming distance* of two codes words = number of positions in which they differ.
  - ✥ **Example**: code = $\{000, 101, 110, 011\}$; each two words have Hamming distance 2.
- *Minimum distance* of a code = least Hamming distance of distinct code words.
  - ✥ **Example**: for any code defined by parity checks, minimum distance = 2.
  - ✥ Aside: a code can correct e errors iff it has minimum distance at least $2e + 1$.

RAID Level 6

To tolerate f simultaneous failures, we use a code C with minimum distance $f+1$.

- Require that corresponding bits of all disks form a code word.
- If f disks fail, we cannot be unsure of what the entire code word was, or else there are two candidates that differ in only the f positions of the failed disks.

Example

The *Hamming code* is a single-error-correcting code (minimum distance = 3) defined by a matrix such as:

$$\begin{array}{ccccccc} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{array}$$

- General rule: n rows, $2^n - 1$ columns.
    - ❖ All columns but the all-0 column.
    - ❖ Columns with one 1 are redundant disks; others are data disk.
- Each row limits the code words; it says there must be an even number of 1's among the positions in which the row has 1.

Computing Redundant Disks

Using the Hamming matrix, we have 16 code words.

- The first four bits (the bits on the data disks) can be chosen arbitrarily.
- But then each of the redundant disks has its bit determined, by rows 1, 2, and 3, respectively.

Example

8-bit blocks, 4 data disks with first blocks:

$$\begin{aligned} x_1 &= 11110000 \\ x_2 &= 10101010 \\ x_3 &= 00111000 \\ x_4 &= 01000001 \end{aligned}$$

Redundant Disks:

$$\begin{aligned} x_5 &= 01100010 \ (x_1 \oplus x_2 \oplus x_3) \\ x_6 &= 00011011 \ (x_1 \oplus x_2 \oplus x_4) \\ x_7 &= 10001001 \ (x_1 \oplus x_3 \oplus x_4) \end{aligned}$$

Handling Two Disk Failures

Suppose disks a and b fail.

1. Find a row r where columns a and b of the Hamming matrix differ.
   - Assume WLOG that a has 1 and b has 0.
2. Use row r to solve for the contents of disk a.
   - Possible because of the disks with 1 in row 1, only a is unknown; it is the modulo-2 sum of the other disks.
3. Now, with only disk b unknown, use any row with 1 in column b to recover disk b.

Example

Suppose disks 2 and 5 fail.

- Row 2 of Hamming matrix has 1 in column 2, 0 in column 5.
- Row 2 tells us $x_2 = x_1 \oplus x_4 \oplus x_6$.
- Next use row 1 to recover $x_5 = x_1 \oplus x_2 \oplus x_3$.

Physical Layout of Data

In relational terms:

- *Field* = sequence of bytes representing the value of an attribute in a tuple.
- *Record* = sequence of bytes divided into fields, representing a tuple.
- *File* = collection of blocks used to hold a relation = set of tuples/records.

In object-oriented terms (ODL):

- Field represents an attribute or relationship.
- Record represents an object.
- File represents extent of a class.

Fields

- Typical attribute can be represented by a fixed length field.
  - Integers, reals $\rightarrow$ 2–8 byte fields.
  - `CHAR(`n`)` $\rightarrow n$ bytes.
  - `VARCHAR(`n`)` $\rightarrow n+1$ bytes.
  - SQL2 date $\rightarrow$ 10 bytes.
- More difficult:
  1. Truly varying character string.
  2. Many-many relationship $=$ set of pointers of arbitrary size.

Fixed-Length Records

1. *Header* = space for information about the record, e.g., record format (pointer to schema), record length, timestamp.

2. Space for each field of the record.

   - Sometimes, it is more efficient to align fields starting at a multiple of 4.

Blocks of Fixed-Length Records

1. *Block header* = space for info such as:

   - Links to other blocks of a data structure.
   - Role info for this block, e.g., for which relation does the block hold tuples?
   - Directory of records in the block.
   - Block ID.
   - Timestamp.

2. Some number of records. Usually, records are not split among blocks, but a *spanned record* can be divided between two blocks.

   - Avoids wasted space.
   - Useful to have each record piece point to the other piece in its header.