CPS352 Lecture: File Structures Last revised March 17, 2021

Objectives:

1. To make clear the performance ramifications of disk seek time.
2. To make clear the notion of blocking on disk.
3. To briefly overview file structures.

Materials:

1. Projectable of memory hierarchy (Figure 12.1)
2. Projectable of hard disk structure
3. Projectable of table of row positions
4. Projectable of reasons for clustering book information
5. Projectable of what cluster might look like

I. Introduction

A. We saw at the beginning of the course that a DBMS allows us to view the data at three levels of abstraction: the view level, the conceptual level, and the physical level. Thus far, we have focussed our attention on the top two levels.

B. We now shift our focus to the physical level. Understanding the physical implementation of higher-level constructs is important for the following reasons:

1. Someone has to write the DBMS software that implements them. (That, however, is not the focus of this course!)

2. Most systems give the database administrator a range of options for the mapping of the data to physical storage. Intelligent use of these options can make a very significant (and user-noticeable) difference in the way the system performs. To "tune" the system properly, the DBA must understand what is happening at the physical level.

3. To some extent, at least, an understanding of what must be done at the physical level to implement a certain construct can influence higher-level design choices. That is, to some extent the ideal of total abstraction between the various levels must be tempered by the realities of system performance.

C. The performance of the DBMS file system is often the determining component of overall performance.

II. Data Storage Hardware

A. The text presents a good overview of current storage technologies, including RAID. The notion of a memory hierarchy is pivotal.

PROJECT Memory Hierarchy - Figure 12.1

We won't spend much time on this in class. (We discuss more fully in CPS311.) But there are a few key points worth noting:

1. Throughout the "database era", three basic kinds of technology have been used for information storage.

a. Primary memory technologies (cache and main memory - today, semiconductor chips) are characterized by

i. Speed for the primary system comparable to the speed of the CPU.

ii. Volatility - when an application terminates (normally or due to a power failure or crash), information in this level of hierarchy ceases to exit.

iii. The CPU only has direct access to this level - therefore information found further down must be copied up to this level, and permanent information (e.g. contents of the database) changed at this level must be copied back down if it is to persist.

b. Secondary memory technologies (today flash memory, disk) are characterized by

   i. Speed for the overall system <u>much</u> slower than primary memory (by a factor of 1,000,000:1 or more for disk).

      While flash memory is much faster than disk (by several orders of magnitude for reads), capacity limits mean that really large databases must be stored on disk (with flash possibly used to cache frequently-accessed data).

      In our this and future classes, we will assume the database is stored on disk - but if it is in flash memory the basic issues are similar, though the speed ratios differ.

   ii. Not volatile. Information in secondary storage remains when an application terminates, though it can be lost due to mechanical failure or - in the case of disk - if a power failures occurs in the middle of a write.

      A loss of information due to catastrophic causes (disk head crash, disk bearing failure, flash failure, fire, etc.) is often total.

c. Magnetic tape and optical disks (sometimes referred to as tertiary memory technologies) can be used for backup and archiving data, but are way too slow to use for the database proper for most purposes. We will not discuss these further.

2. Capacities of both primary and secondary memory systems have grown dramatically. Today's systems easily have 10's of 1000's of times more of each than systems at the start of the database era. But speeds of disk, however, have changed very little, and are not expected to change significantly in the future either.

3. The large speed gap between primary and secondary technologies (especially disk) implies that, for all practical purposes, time to access information in secondary storage is the major determining factor in overall system performance.

B. Disk speeds are dominated by access time.

1. Access time includes the time needed to position the disk head o the correct track, plus the time needed for the desired information to rotate so that it is starting to pass under the head. Access times on the order of 10 ms are typical. (We will use this number for the sake of discussion)

   PROJECT: Hard disk structure

   This is a critical reality - a disk access takes time orders of magnitude greater than computational times. E.G. if a CPU executes 1 billion instructions per second, then a disk access takes the same time as 10 million instructions. **(**Compare time of one second of class to total Gordon education = 60 sec/min x 60 min/class x 42 classes/course x 47 courses to graduate = about 7 million!

2. Typically about 1% of the time is actually spent transferring the information - the rest is access time. For this reason, information in secondary storage is kept in fixed size units (called pages or blocks) - typically 4K or 8K bytes. Any access to information is done a whole block at a time - so whenever data is in the database is accessed, the whole block containing the data is copied between main memory and secondary storage. Thus, transferring a whole block of data actually takes only very slightly more time than transferring a single byte.

   This is another key idea - a major consideration in planning strategies for processing queries is minimizing the number of disk accesses needed.

3. Actually, two considerations help to keep the number of time-consuming seeks down.

   a. It is necessary to allocate one memory buffer for each table. If the buffer for a table already contains the block we need, we don't need to access the disk again.

      If more memory is available for buffers, we can further reduce the need for disk accesses by buffering multiple blocks

   b. When transferring data from/to physically contiguous blocks on the disk, only one seek is often needed (to get to the first block) - so the time taken to transfer several contiguous blocks is little more than the time needed to transfer one.

C. Therefore, a major goal of the design of DBMS file systems (arguably <u>the</u> major goal) is to minimize the number of disk accesses necessary. There are three ways this is done.

1. Keeping information that is needed for a particular operation together in a single block on disk, or in contiguous blocks, thus minimizing the number of separate accesses needed.

2. Keeping copies of recently-used information in buffers in primary memory, so that if the same information is needed again if can be accessed without having to go to the disk again.

   Example: Consider a SQL query like

   ```
   select avg(salary)
   from employees
   where department = 'Software';
   ```

   PROJECT

   Suppose the employees table were stored in a disk file that occupied 1000 disk blocks. To process this query, the system might need to read each disk block and examine each row in the block to see if the department value were

'Software'. In all likelihood, the overall time to answer the query would be dominated by the disk access time - about 1000 * 10 ms = 10 sec.

Now suppose we did a second, similar query

```
select avg(salary)
from employees
where department = 'Hardware';
```

If buffering were not used, this query would require the same amount of the time to satisfy as before - i.e. the total time for the two queries would be 20 seconds. However, if a copy of the data read from disk were kept in buffers in primary memory, then the second query could be answered using this data without any need to go to disk at all. The time needed for the second query would be dominated by processing time which, though not negligible, would still probably be much less than 1 second) - so the the total time for the two queries would be little more than the time for the first.

3. Parallelism - spreading information across multiple disks, so that several disks can be going through the physical operations needed to access information at the same time.

   Example: suppose a certain operation needed to access 1000 blocks on disk. The total time would be on the order of 1000 * 10 ms = 10 seconds. But if the information were spread over 10 disks, such that 100 blocks were on each, and the disks could be accessed in parallel, it might be possible to do the operation in about 1 sec.

D. For database server systems, it is quite common to find some sort of RAID configuration being used. RAID means "Redundant Array of Independent/ Inexpensive Disks".

1. As the book discussed, there are a number of different configurations (known as RAID levels) possible - though really only a few that are widely used.

2. RAID systems may seek to improve throughput by a technique known as <u>striping</u>, in which a single file is spread over multiple disks. Thus, multiple accesses to different parts of the same file can often be performed in parallel (assuming that the parts being accessed are on different disks).

3. RAID systems may seek to improve reliability by <u>replication</u> of data, so that if a disk fails, the data it contained is available somewhere else.

   This becomes increasingly important as the number of disks involved increases. While a single disk is very reliable, if one has a large number of disks the probability that some one will fail increases.

   Example: If a single disk has a MTBF of 50,000 hours, then one would expect one failure every 5 years. But if a system had 1000 of these disks, then one would expect about 16 failures every month!

III.DBMS File Structure

A. The database itself is stored in the form of a file or files on one or more disks. It is at this level that the DBMS interfaces with the host operating system which ultimately "owns" the disks.

   1. Some DBMS's store their data in a collection of files - perhaps one for each higher-level entity (relation, class etc.) plus additional files needed by the implementation.

      Example: mySQL: a database is represented by a Unix directory, and each table is stored in a Unix file whose name is the same as the name of the table.

      SHOW

      ```
      ssh to joshua and su to root
      ls /var/lib/mysql
      ls /var/lib/library
      ```

(Note that access to the database directory is prohibited for ordinary users (only access is possible through DBMS which "owns" the files), though root can access, of course.)

2. Other DBMS's allocate a single, very large file from the host operating system and then build their own file system within it.

   Example: Db2 stores each database within one or more large files on the disk. Each file may contain any number of tables, indices etc. The DBMS manages their arrangement within the file.

   SHOW

   ```
   ssh to joshua and su to root
   cd ~db2inst1/db2inst1/NODE0000/
   ls - note how each database has own directory
   ls -l SAMPLE/T0000000
   more SAMPLE/T0000000/C0000000.CAT
   ```

B. The data stored on disk is of several kinds:

1. The actual user data that the system users create and manipulate.

2. Various system data:

   a. The data dictionary

   b. User access control information - names of authorized users and the specific data access they are allowed either as the creator of a protected object or as a result of a `grant` statement.

      Some systems (such as mysql) also store user passwords for the database that are separate from the system passwords used to log onto the host operating system. In this way, several different users with different system login accounts can share the same database or vice-versa.

(But note that Db2 as we have installed it uses the system login accounts to also manage database access.)

c. Indices to the various tables - more about this in the next topic

d. Statistical data which facilitates efficient query processing - more about this later.

e. Log data used for Crash recovery - more about this later.

3. In this lecture we focus on the way the database tables are stored.

C. Storing Relational Tables

1. Storing Individual Rows

a. Typically, each row of a table is stored as a contiguous series of bytes somewhere on the disk. And typically, the successive rows of a table are stored contiguously one after another.

There is an alternative approach known as column-oriented storage, where all the values of each column in successive rows are stored together - but we will not pursue this.

b. The simplest case arises when all the attributes of the table have a fixed size. Then the storage required is the sum of the space needed for the attributes plus one or more bytes of bit vector to allow for specifying that a particular attribute is null - in which case the actual value stored for the attribute is ignored.

Example: As a result of the following create table statement (constraints removed for clarity):

```
create table employee (
   ssn char(11),
   last_name char(20),
   first_name char(20),
   salary integer,
   supervisor_ssn char(11)
);
```

PROJECT

Since each character is typically stored as a single byte, and an integer is stored as 4 bytes, and 1 byte is needed as a bit vector to specify nulls since there are only 5 attributes, an employee row might be stored as 11 + 20 + 20 + 4 + 11 + 1 = 67 bytes.

c. If the table contains a `blob` or `clob` object, often the actual value of the large object will be stored elsewhere and the row itself will just contain a few bytes of pointer to it. (In this case, the space needed for the actual row doesn't vary)

d. If the table contains an attribute like `varchar` or one that is list valued, the number of bytes needed to store different rows vary.

   This creates a lot of complexity, so some DBMS's achieve a compromise between tables with fixed length rows and tables with variable length records as follows: The main storage structure for a table is a sequence of fixed length rows, containing the fixed-length attributes plus a fixed amount of size information plus a pointer to another location which contains the variable-length attributes.

e. For simplicity, we'll assume a situation in which the space needed to store each row in a table is the same, and won't worry about handling of variable length rows or large objects.

2. In general, the number of bytes needed for a table row will be much smaller than the block size (4K or 8K), so each block can hold multiple

rows from the same table. (The reverse case - if a row is too big for one block - is one we won't consider because it would be very rare - though it would have to be dealt with somehow).

a. Of course, the number of bytes in a block (often 4096 or 8192) is generally not an even multiple of the row size. For example, for the employee table discussed above, a block size of 4096 would allow 4096/67 = 61.13 rows to fit in a block. Because allowing rows to span block boundaries would introduce all kinds of complications, this is rarely allowed - so we could put 61 rows in a block and leave 9 bytes in each block unused.

b. So, for a given table, we would calculate row_size and rows_per_block, where row_size * rows_per_block is as close as possible to the block size (which we will call block_size) without exceeding it.

c. A table will occupy as many blocks as necessary to store all the rows of the table. It is customary, in this context, to treat the rows as if they were numbered 1, 2, 3 ... (no row 0), and to treat the blocks as numbered 1, 2, 3 ... (no block 0), so we will have row r stored in block b calculated from the formula:

   ```
   b = 1 + (r-1) / rows_per_block  - where / is integer
   division (discard remainder),
   ```

   and the offset of the start of row r from the beginning of block b is calculated by

   ```
   offset = ((r-1) % rows_per_block) * bytes_per_row
   ```

   So, for our example of the employee table with 67 bytes per row and 61 rows per block we get the following

```
row  block    offset in block

1    1        0
2    1        67
...
61   1        4020
62   2        0
```

PROJECT

3. Another issue that arises is how the rows are arranged in the file.

    a. Keeping the rows of the table in some order appear to not be practical, since it would require that we moved all the rows around to insert a new row in the middle of the other rows - which could entail moving thousands of rows.

    b. One approach is to not worry about the order. Remember that a relation is a set, so it does not have inherent order. We just add rows to the end of the table in the order added; if we delete a row in the middle of the table the space can be reclaimed for another row that is added instead of trying to move rows down to squeeze out the vacant space.

    c. There is a structure know as a B-Tree - which is considered in CPS222 and we will look at in the next topic - which actually does allow us to keep the rows of the table in some order based on the value of some attribute.

        Of course, any given file can only be stored in a sorted order based on one key.

        We will discuss this further in the next topic.

D. Storing all the rows of a given table together (in their own file or in a region within a file) however, is not always the best solution.

1. Frequently, it is the case that related rows from two or more different tables are frequently retrieved together.

   In the process of normalization, we often decompose a relation scheme into two or more relation schemes in order to achieve BCNF or 4NF. However, some retrievals will want most or all of the data that existed in the original scheme, requiring a join, so we will want to include the original scheme as a view, constructed upon demand by natural join. It is quite possible that we will find that most of the retrievals of the data are done through the view, rather than through the individual conceptual schemes, requiring computationally expensive joins for each query.

2. In that case, there is something to be gained by mixing records from two or more different relations in the same file.

   Example: Consider the book table in our library example

   ```
   Book(call_number, copy_number, barcode, title, author
   ```

   with dependencies:

   ```
   barcode -> call_number, copy_number
   call_number, copy_number -> barcode
   call_number -> title
   call_number ->> author
   ```

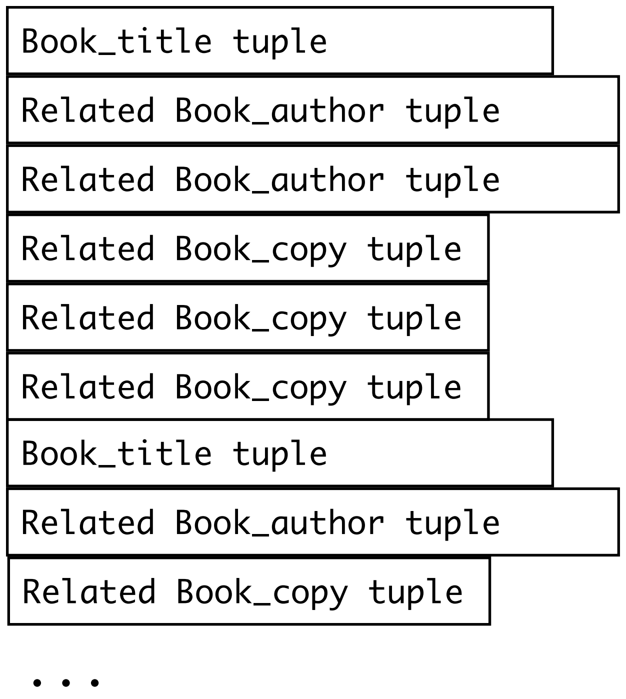
   To achieve 4NF, we must decompose this into:

   ```
   Book_title(call_number, title)
   Book_author(call_number, author)
   Book_copy(call_number, copy_number, barcode)
   ```

   PROJECT

   But now retrieving full information on a book given, say, its call_number and copy_number requires two joins. If each relation is stored on disk separately, this requires a minimum of three disk accesses - and probably many more (because of needing to search for the matching rows.)

However, there is a sort of "parent-child" relationship here. For each Book_title tuple, there will be one or more Book_author tuples and one or more Book_copy tuples. But each Book_author and Book_copy tuple will be associated with one and only one Book_title tuple. Thus, we might choose to put all the Book_author and Book tuples associated with a given Book_title tuple in the same disk block as the Book_title tuple, so we can retrieve all the information about the book with just one disk access. The way this is stored in the file might look like this:



PROJECT

3. Some DBMS's (but not Db2) define a `CREATE CLUSTER` statement and allow the CREATE TABLE statement to place a table in a cluster. Exactly how this is implemented is implementation specific.

4. Unfortunately, the term "cluster" is used for a variety of things. In fact, we will also meet the term `cluster` in connection with the index topic we look at next, but with a different meaning.