CPS352 Lecture: File and System Organization Last revised 2/21/2017

Materials:

1. Projectable of memory hierarchy (Figure 10.1)
2. Projectable of Korth and Silberschatz (2nd ed) p.217
3. Projectdable of RDB cluster demo

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.

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 key component of overall performance. There are two attributes that can be optimized:

1. Response time - defined as the time between the issuance of a command and the time that output for the command begins to be available. (E.g. if the command is a select statement, the time until the first row of the result appears.) Of course, we want to minimize this.

2. Throughput - the number of operations that can be completed per unit time. Of course, we want to maximize this.

3. For single-user DBMS's, response time is what needs to be optimized. A human user wants minimal possible delay between issuing a command and seeing results.

4. For a database server for a web site being accessed by many users, throughput is often what needs to be optimized. If, for example, a system can only process 10 transactions per second, but on the average 11 users attempt an operation every second, the site will soon become badly backlogged!

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

PROJECT Memory Hierarchy

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”, two basic kinds of technology have been used for information storage.

a. Primary memory technologies (today, semiconductor chips) are characterized by

(1) Speed for the overall system comparable to the speed of the CPU.

(2) Volatility - when an application terminates (normally or due to a power failure or crash), information is lost

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

(1) Speed for the overall system _much_ 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 primarily stored on disk (with flash possibly used to cache frequently-accessed data)

(2) Not volatile. Information in secondary storage remains when an application terminates, though it can be lost due to mechanical failure, disaster such as fire, and information currently being written can be lost in a power failure. A loss of information due to catastrophic causes (head crash, bearing failure, fire, etc.) is often total.

2. 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.

3. 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.

4. Speeds of disk, however, have changed very little, and are not expected to change significantly in the future either. The large speed gap between primary and secondary technologies (especially disk0 remains the key issue in DBMS performance. For all practical purposes, time to access information in secondary storage is the major determining factor in overall system performance.

5. Disk speeds are dominated by access time.

   a. Access time includes the time needed to position the disk head to 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.

   b. Typically about 1% of the time is actually spent transferring the information - the rest is access time.

   c. For this reason, information on disk is always organized in blocks - relatively large chunks (e.g. 4K or so) of contiguous information that is read/written as a unit. A system never reads or writes a single disk byte - it always reads or writes the whole block containing a desired piece of information.

6. NAND flash memory also reads and writes data in blocks, though the read access time is much faster (by about 4 orders of magnitude) though the overall write time is much slower.

7. A major goal of the design of DBMS file systems (arguably _the_ major goal) is to minimize the time spent waiting for disk accesses. There are three ways this is done.

   a. Keeping information that is needed for a particular operation

together in a single block on disk, thus minimizing the number of separate accesses needed.

We will see an example of how this might be done later.

This can help both minimize response time and maximize throughput.

b. Keeping copies of recently-used information in buffers in faster memory (flash or primary), 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';
```

Suppose the employees table were stored in a disk file that occupied 1000 disk blocks - all on one disk - random or sorted in some order other than department. To process this query, the system would 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 max(salary)
        from employees
        where department = 'Sales';
```

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 flash or main 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 be much less than 10 seconds (probably much less than 1 second) - so the the total time for the two queries would be little more than the time for the first.

This can help both minimize response time and maximize throughput.

c. 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. If the information were spread over 10 disks, such that 100 blocks were on each, it might be possible to do the operation in just 1 sec.

Though this does not help response time, it can greatly improve throughput.

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

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

b. RAID systems may seek to improve throughput by a technique known as striping, 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).

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

i. 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!

ii. Redundancy can also be used to improve throughput for reads - if there are multiple copies of an item, then any copy can be read.

iii. Redundancy creates an issue on write though - since all copies must be updated.

E. An earlier edition of our text included an overview diagram showing the overall system structure for a typical, full-feature DBMS. (Less sophisticated DBMS's may omit several of the components shown):

PROJECT - page 217 (2nd ed)

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

a. 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 `/var/lib/mysql` and `library` subdirectory on joshua.

Note that access to database directory is prohibited for ordinary users (only access is possible through DBMS), though root can access, of course

b. 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.

SHOW
```
ls ~db2inst1/db2inst1/NODE0000/
ls ~db2inst1/db2inst1/NODE0000/SAMPLE
ls -l  ~db2inst1/db2inst1/NODE0000/T0000000
```
DO more on file `C0000000.CAT`

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

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

b. Various system data:

i. The data dictionary which maps data names used by users (for tables, columns, etc.) to the actual stored data, and which contains information about data types, constraints etc.

ii. User access control information - names of authorized users and the specific data access they are allowed. Some systems 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 can share the same host system login account, but have different access to the database, or vice-versa.

iii. Statistical data about the frequency of various kinds of accesses to the data, and about the frequency of occurrence of various data values. This data can be used to help tune the system for better performance by making frequently-performed operations more efficient, and can help make queries more efficient.

iv. Some systems store system data in a way that is different from user data. Others, however, treat the system data as structures in the database just like any others, but with access only allowed to the DBA.

Example: mysql stores system data in various tables in a database called mysql - only accessible to the DBA.

login to mysql on jonah as root;

```
use mysql;
show tables;
describe user;
```

Example, db2 stores system data in a special form, but defines various SYSTEM VIEWS which allow the DBA to access (and sometimes change) the system data using SQL as if it were an ordinary table. There is separate system data for each database in the instance.

```
su - db2inst1
db2 -t
connect to sample;
list tables for all;
select tabname from syscat.tables
    where tabschema not like 'SYS%'
```

c. Indices to the various data files stored on the disk. If a certain attribute is often used as the basis for selecting records within a relation, then an index to the relation based on that attribute can speed queries immensely.

d. Log data that maintains a historical record of changes to the database. This is useful for:

i. Crash recovery.
ii. System auditing: to catch unauthorized changes.

Note: In the diagram this is separated from the other data shown as disk storage. This is because this data is sometimes stored on a separate medium - but often necessity dictates that it, too, is stored on the disk.

3. Because access to the disk is relatively slow (compared to internal processing speeds), the system maintains a pool of main-memory buffers to contain data which has been fetched from the disk and which will be needed soon.

a. As a bare minimum, one buffer is needed to store the most recently accessed block from the disk for each relation.

b. Typically, a pool of buffers is used to retain data in memory that has been used once and is likely to be used again soon - thus

saving a second disk access the next time the data is needed.

c. Since the available buffer space is finite (and often much smaller than the database), the DBMS must include a software component to manage the pool of buffer space most efficiently.

4. The DBMS software must include a query parser to accept and "understand" user queries. Associated with this is a strategy selector that "plans" the strategy for carrying out the query. The sophistication of this parser/strategy selector can vary widely:

a. Parsers for a query language like SQL are relatively simple, while a language like QBE is more sophisticated. Some newer systems include "natural-language" front-ends based on AI techniques that can be very complex.

b. Strategy selectors can range from nonexistent to very sophisticated.

Example: Given schemes R1(A,B,C)
R2(C,D,E)

with indices for the C attribute in each tuple and for the E attribute in R2

and query

$$\prod_{A,D} \sigma_{E='BOSTON'} (R1 \bowtie R2)$$

An unsophisticated strategy would be to compute the natural join of R1 and R2 by scanning through R1. For each tuple in R1, the C index to R2 could be used to find the matching tuple. Then, after the join has been completed, the next

step would be to examine each resulting tuple to see if its "E" attribute = 'BOSTON', then construct new tuples for those selected containing only the A and D attributes.

A more sophisticated strategy would first examine relation R2, selecting only tuples where E = 'BOSTON' (by using the index on the E attribute). Then it would only be necessary to look up the corresponding R1 tuple (using the C index), then construct a new tuple consisting of R1.A, C, and R2.D.

Even more sophisticated strategies might make use of data about the size of the two relations and the frequency of occurrence of the various values. For example, if the R1 relation contains only 10 tuples and R2 contains 10000, of which 1000 have the E value = 'BOSTON', then the first "unsophisticated" strategy would actually be the most sophisticated. A good strategy selector would take this statistical information into consideration in making its choices.

A sophisticated strategy selector can make a very significant difference in overall system performance - at the cost of increased software size and purchase price.

5. Sophisticated DBMS's include crash recovery facilities that allow the database to be restored to a consistent state after a system crash due to power failure, hardware failure, or software failure. This is done by maintaining a log of changes to the database, which can be used to restore the system when it is started up again. (Unsophisticated systems - such as many microcomputer DBMS's - totally lack such facilities.)

6. DBMS's that allow multiple users concurrent access to the data need software components to prevent inconsistencies resulting from

multiple simultaneous modifications of the same item.

F. In this lecture we focus on the file and buffer components of the system. Then we move on to consider query parsing/strategy selection, crash recovery, and concurrency control in turn in subsequent lectures. For the most part, we will couch our discussion in terms of the relational model.

II. File structure considerations

A. The file structure of a database system can have a very significant impact on system performance. This is because disk accesses are orders-of-magnitude slower than internal processing, as we have noted.

B. The simplest way to physically implement a conceptual database is as a collection of files - one per relation/record-type, or as a group of separate regions within a single large file.

1. Within a given file (or region within a file) the records will therefore generally be of a fixed size, dictated by the size of a row in a particular table. Any given block in the file can hold

   `records_per_block = record_size / block_size` (using integer division, so there is generally some wasted space since records are not allowed to span blocks)

   Any record in the file can be located by

   ```
   block_number = record_number / records_per_block
   offset_in_block = record_number % records_per_block * record_size
   ```

2. Variable length records may be needed in some cases

   a. While the relational model demands that records be normalized

(no repeating groups), the other models do not. A record type with repeating groups requires either that we allocate a fixed space for each record (able to hold the maximum number of repetitions of the group) or else we must use variable length records.

b. Some database applications demand the one or more fields of a record be variable length. For example, a medical records database may want to include the doctor's observations, diagnosis, and recommendations for each patient visit. For some patients, this may be very short - e.g. "Patient coughs - common cold - take cough syrup and go to bed.". For others, the writeup may be a full page of text. Obviously, if some field in the record is variable length, then the records must be.

c. Variable-length records generally complicate life, however:

i. Locating records becomes harder. because the number in a block can vary.

ii. When a fixed length record is deleted, the space reclaimed can be used for another record easily, since all are of the same size. Not so with variable length.

d. Some DBMS's achieve a compromise between fixed length and variable length records as follows: The main storage structure is a file of fixed length records, containing the fixed-length fields plus pointers to another file which contains the variable-length fields.

e. In the case of multimedia databases, large "values" such as movies are stored as "blobs" (binary large objects) in separate files. The same approach is used for large chunks of text (e.g. a chapter of a book) which may be stored separately as a

"clob" (character large object.) Again, the blob and clob values can be stored in separate files referenced from the main file.

3. One issue that arises is how the records are arranged in the file. Ideally, we would like to put records together in such a way as to minimize accesses. Unfortunately, no one arrangement will optimize all operations - instead, we must base the arrangement on the operations we think are most likely to be performed (or we know are most likely based on past history recorded by the DBMS).

   Example: Suppose a certain file is often processed sequentially in ascending order of some field (e.g. last name.) Then it makes sense to store the records in that order, since all of the records in any one block will be processed at about the same time, so we only need to get each block once.

   Example: Suppose a personnel file has a primary key of SSN, and also contains a field giving employee last name. Normal file design would suggest that the file be kept in ascending order of the primary key. However, if most queries are made by name rather than SSN, and if duplicate last names occur with some frequency, then it makes sense to organize the file so that all of the employees with the same last name are in the same block.

4. Of course, maintaining a file in a particular sorted order can facilitate queries, but it makes record insertion (and to some extent deletion) much more costly, since records have to be moved.

   a. One approach is the use of buckets: the file is logically divided into a set of buckets (each of which may be several blocks), and all records with the same key value (or range of key values if the number of occurrences of one value is small) are placed in the same bucket.

      i. Generally, all buckets are of the same size, though this is

not always necessary.

ii. Initially, the file is configured so that each bucket has extra room in it. This makes insertions relatively cheap, while also facilitating retrievals. (Of course, if a bucket gets to be full eventually, then it must be split some how.)

iii. This approach trades storage space for processing time - a familiar tradeoff.

b. Another approach is the use of structures such as B+ Trees etc. These were discussed in CPS212 and will come up again in our discussion of indices.

5. Of course, any given file can only be stored in a sorted order based on one key. (The exception would be the case of two keys where one is a prefix of the other - e.g. last_name+first_name, last_name.

6. What control the DBA has over the order of placement of rows in a table is, of course, very DBMS-specific.

C. Storing all the records of a given 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 records from two different record types are retrieved together.

Example: in the process of normalization, we often decompose a relation scheme into two or more relation schemes in order to achieve a higher normal form. However, some retrievals will want most or all of the data that existed in the original scheme, so we will want to include this 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, which would require computationally expensive joins for each query.

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

   Example: Consider the relation

   ```
   Book(call_number, copy_number, accession_number, title, author)
   ```

   with dependencies:

   ```
   accession_number -> call_number, copy_number
   call_number, copy_number -> accession_number
   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(call_number, copy_number, accession_number)
   ```

   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 tuples. But each Book_author and Book 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.

3. As an illustration of a commercial version of this, at one point in this course we were using a DEC product called RDB in this course, which allowed intermingling rows from different tables as described above. (DB2 doesn't seem to make this an option.) The following RDB SQL commands would create a database containing these three relations and would cluster them as described. (The example is not complete; to get maximum efficiency it would be necessary to specify some size parameters for the book_cluster storage area. We omit these to focus on the concept, not the details.)

   PROJECT

III. Buffer Management

A. Thus far, we have talked about the organization of data on disk. When data is transferred to/from disk, it is transferred in units of one or more blocks, and is transferred to/from a main memory location known as a buffer.

B. We have seen that DBMS's typically maintain a POOL of buffers so that a block that has been read from disk that is going to be needed again in the near future can be retained in memory until that time - thus avoiding a second disk access. Since the size of the pool is typically much smaller than the overall size of the database, proper management of this pool can be one of the largest factors in the overall performance of the DBMS.

C. Within the constraints of considerations imposed by crash recovery mechanisms we will discuss later, the buffer manager must make choices as to which buffers to retain whenever all buffers are

in use and a new buffer is needed to satisfy a fetch request (which forces the contents of some buffer to be tossed.)

1. The OPTIMAL policy is to reuse the buffer whose current contents will not be needed again for the longest time into the future.

   a. A similar problem arises in the context of operating systems, where we quickly conclude that the optimal policy is unachievable because we do not know the future.

   b. However, in the context of processing a given query, it is may be possible to achieve optimal replacement by interaction between the query processor and the buffer manager. (Cf the pin and toss primitives above - but additional information can be used too.)

2. Sometimes optimal replacement can be achieved (or approximated well) by using one of the two policies LRU and MRU.

   a. LRU says that when a buffer is needed to fulfill a fetch request, choose the one whose contents have been used LEAST RECENTLY. The assumption is that the past is key to the future, so that a buffer that has not been used for a while is not likely to be used again soon.

   b. MRU is just the opposite. It says that when a buffer is needed to fulfill a fetch request, choose the one whose contents have been used MOST RECENTLY. This will be the right decision in cases where we are cycling repeatedly through all the blocks of a relation, but the relation is so big that we cannot buffer all of it in memory. In this case, the block we have just finished using will not be needed again until all the other blocks of the relation have been cycled through (provided we pin it while we are processing the records it contains.)

3. Sometimes, optimal replacement can be approximated by taking into consideration FREQUENCY of use. Some blocks are used often enough that they are worth buffering in memory, even if the time of their next use cannot be predicted precisely - e.g.

   a. Blocks forming part of the data dictionary.

   b. Root blocks of index structures.

   etc.

4. Of course, many other policies are possible. In practice, however, many systems simply settle for a simple (but generally good) scheme like LRU.