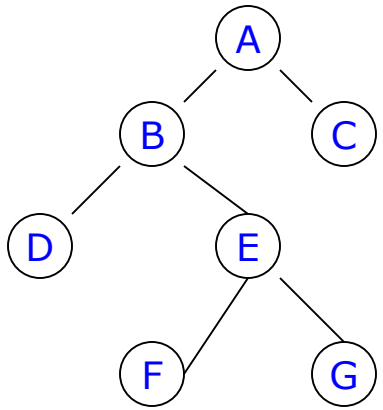
Transaction Management

- Tree Protocols
  - In many instances, the set of items accessed by a transaction can be viewed naturally as a tree or forest
  - E.g., items are nodes in a B-tree; items have different granularities (relations, tuples, attributes).
  - Two different policies may be followed:
  1. each node in the tree is locked independently of its descendants
  2. a lock on an item implies a lock on all of its descendants
  - The latter policy saves time by avoiding locking many items separately
  - E.g., when the content of an entire relation needs to be read, the relation can be locked in one step instead of locking each tuple individually

Transaction Management

- Tree Protocol #1 (TP1)
- Definition: A transaction obeys the TP1 policy if:
  - except for the first item locked, no item can be locked unless the transaction holds a lock on the item's parent
  - no item is ever locked twice by a transaction
- A schedule obeys the TP1 policy if every transaction in the schedule obeys it.
- Example: Consider the following hierarchy of items



Transaction Management

- The following schedule obeys TP1

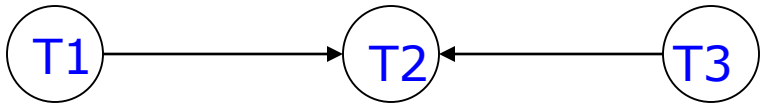
T_1	$L_1(A)$	$L_1(B)$	$L_1(D)$	$U_1(B)$		$L_1(C)$		$U_1(D)$	
T_2					$L_2(B)$				
T_3							$L_3(E)$		$L_3(F)$

T_1	$U_1(A)$		$U_1(C)$						
T_2					$L_2(E)$		$U_2(B)$		$U_2(E)$
T_3		$L_3(G)$		$U_3(E)$		$U_3(F)$		$U_3(G)$	

  - Does it obey 2PL?

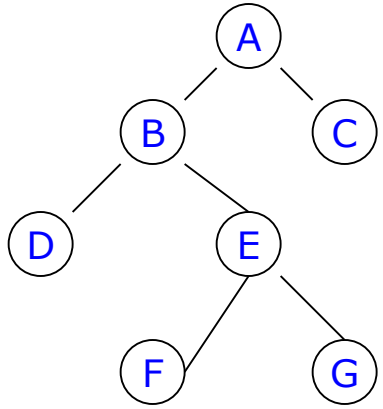
Transaction Management

- Note: A transaction that obeys TP1 need not necessarily obey 2PL.
- Theorem: Every legal schedule that obeys the protocol TP1 is serializable
- Example: The schedule of the previous example is serializable.
  - its precedence graph is acyclic



Transaction Management

- Tree Protocol #2 (TP2)
- Definition: A transaction obeys the TP2 policy if whenever an item is locked, all its descendants are locked
- Indiscriminate locking under TP2 may result in schedules where two transactions hold a lock on the same item at the same time.
- Example: in the hierarchy



transaction T_1 locks E (therefore F,G). Then T_2 locks B, therefore acquires conflicting locks on E,F,G.

Transaction Management

- To avoid conflicts of this sort, the warning protocol may be followed:
  - a transaction cannot place a lock on an item unless it first places a warning on all its ancestors
  - a warning on an item X prevents any other transaction from locking X, but does not prevent them from also placing a warning on X, or from locking some descendant of X that does not have a warning
- Definition: A transaction obeys the warning protocol if:
  1. It begins by placing a lock or warning at the root
  2. It does not place a lock or warning on an item unless it holds a warning on its parent.
  3. It does not remove a lock or warning unless it holds no lock or warnings on its children

Transaction Management

4. It obeys 2PL in the sense that all unlock operations follow all warnings or lock operations

- This protocol acts in conjunction with the assumption that a lock can be placed on an item only if no other transaction has a lock or warning on the same item.
- Furthermore, a warning can be placed on an item as long as not other transaction has a lock on the item.
- Theorem: Legal schedules obeying the warning protocol are serializable.

Physical Data Organization

- Databases may be too large to fit in main memory
- The efficiency of operations on databases residing in secondary storage relies heavily on the availability of good storage organization techniques
- Performance measures include the time required to perform database operations such as selections or joins
  - E.g., selections should take time proportional to the number of tuples retrieved rather than the size of the relation
- Storage organization techniques include:
  - Heaps
  - Hashing
  - Indexed sequential access methods
  - B-trees, B^+-trees

The Physical Data Model

- Records, Fields and Files
  - At the conceptual or logical level, a database is a collection of entities and relationships, or a collection of tables representing entities and relationships.
  - At the physical level, a database is a stored collection of records, each consisting of one or more fields.
  - Fields contain values of elementary data types (e.g., integer, real)
  - A record is used to physically store each of the basic objects at the conceptual level
  - E.g., a tuple of a relation can be stored as a record with each component of the tuple stored in one field.
  - Records can be viewed as instances of a record scheme.

The Physical Data Model

- A database contains collections of records with the same number of fields, with corresponding fields having the same name and data type.
- The list of field names and their corresponding data types is called the format of the record.
- A file is a collection of records with the same format. For instance, a file physically represents a relation.
- Two-level storage
  - The physical store where records and files reside can be thought of as an array of bytes numbered sequentially.
  - Files normally reside in secondary storage. In order to perform operations on data found in file records, records must be moved from secondary storage to main memory. Once they are moved operations can be performed very fast compared to the speed of data transfer between main and secondary storage.

The Physical Data Model

- Blocks
  - Secondary storage is partitioned in segments (blocks) of a substantial number of bytes (2^9-2^{12}) with several records in a block
  - Transfer of data occurs in units of a block.
  - The cost of database operations depends on the number of blocks moved between main memory and secondary storage.
  - The efficiency of operations improves when records of a file lie within the same block or a relatively small number of blocks.
- Cost of DB Access
  - The unit cost of db operations is the block access, i.e., the time required for reading from or writing to a single block.
  - This cost of performing main memory operations is negligible compared to the cost of block transfer.

The Physical Data Model

- With the buffering of blocks in main memory we can avoid having to transfer a block from secondary storage to main memory.
- The time to access a block in secondary storage depends on the location where the last block access took place.

- Simplifying assumptions:
  - There is a fixed probability that a block will need to be transferred.
  - The cost of a block access does not depend on what accesses were made previously.
  - Each block access costs the same.

The Physical Data Model

- Pointers
  - A pointer to a record r is data sufficient to locate r efficiently.
  - A pointer can be of different types such as, e.g., the absolute address of the beginning of record r.
  - Absolute addresses are often undesirable: we might permit records to move around within a block or group of blocks. If absolute addresses are used as pointers and record r is moved, all pointers to r must be updated.
  - A pair (b,k) can be used as a pointer, where b is the block in which r resides and k is the value of the key for r.

The Physical Data Model

- Pointers
  - To locate a record r given a pair (b,k) it is sufficient to know that:
    - All records in block b have the same format as r; therefore none can agree with r in its key fields.
    - The beginnings of all the records in block b can be found.
    - Each record in block b can be decoded into its field values given the beginning of the record.
- Pinned Records
  - When records may have pointers to them from unknown locations, we say that the records are pinned.
  - If records are unpinned, they can be moved around within a block or from block to block.

The Physical Data Model

- Pinned Records
  - When records are pinned, they cannot be moved if the pointers to them are absolute addresses. They can only be moved within the same block if a pair (b,k) is used as a pointer.
  - Records cannot be deleted if they are pinned: if there exists a pointer p to a record r and r is deleted, then if at some later point in time a record r' is put in the same place as r, pointer p will point to r'. Pointer p is called a dangling pointer.
  - Even in the case where block-key pairs are used, the problem cannot be avoided: if r' has the same key as r, we would still have an unintended reference to a record.
  - To avoid this problem, each record includes a bit (called deleted bit) which is set to 1 if the record is deleted. If we reach the record by following a pointer, we will know if it is deleted or not.

The Physical Data Model

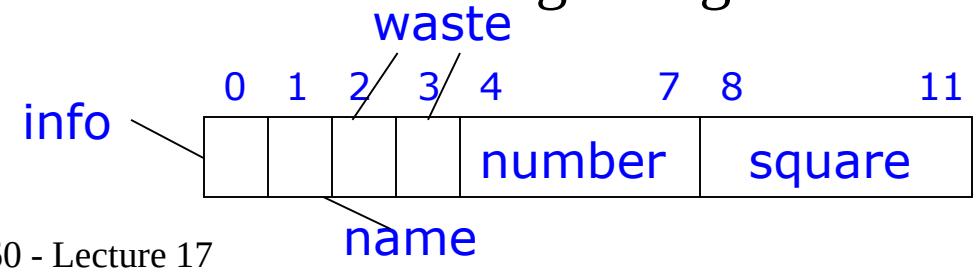
- Record Organization
  - The fields in a record must be arranged in such a way that their values can be accessed. If all fields have fixed length, then only an order must be chosen: each field starts at a fixed number of bytes (offset) from the beginning of the record.
  - Once a record is located, the field can be found by moving forward a number of bytes equal to the offset for that field
  - Additional bytes may be required for each record:
    - Some bytes denote the format of the record. For instance if the record belongs to more than one relation, we may wish to store a code indicating the relation of each record. Alternatively, we can store only one type of records in any block and let the block indicate the type of its records.

The Physical Data Model

- Additional bytes may be required for each record:
  - One or more bytes may denote the length of the record. If the record involves only fixed-length fields, then the length is implicitly derived.
  - A byte including a deleted bit and a used/unused bit: the latter is needed when blocks are divided in areas, each of which holds a record. This bit indicates if there actually exists a record or if it is empty space.
  - Waste space: useless bytes may be added so that the bytes where the records begin have convenient addresses.

The Physical Data Model

- Example: record type: NUMBER
  - Fields:
    - Number: integer (key field) – always holds a positive integer
    - Name: a single byte indicating the first letter of the English name for the number
    - Square: integer – holds the square of the number
  - If an integer takes 4 bytes, a total of 9 bytes is required for the record. In addition, a deleted bit and a used/unused bit is needed. Waste space is needed if integers are required to be stored in records whose beginning address is a multiple of 4.



The Physical Data Model

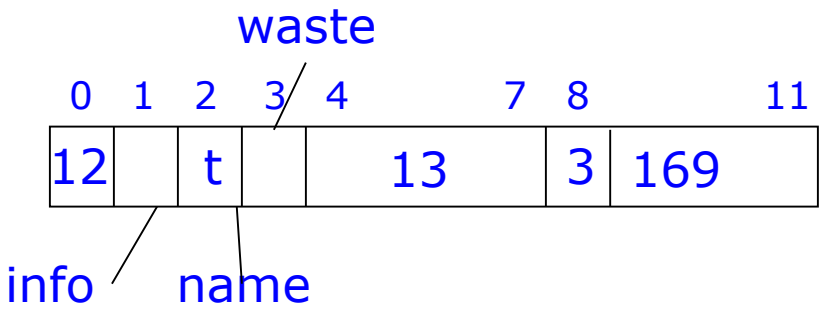
- Variable-length records
  - Record formatting problems occur when fields are allowed to vary in length: fields do not begin at the same offset for every record of a particular format
  - Record formatting strategies:
    - Attach a count at the beginning of each field. Its value is the number of bytes the field occupies. Although redundant, the total length of the entire record is also stored in the beginning of the record.
    - Place, in the beginning of each record, pointers to the beginning of each variable length field and a pointer to the end of the last field. All fixed-length fields precede all variable-length fields.

The Physical Data Model

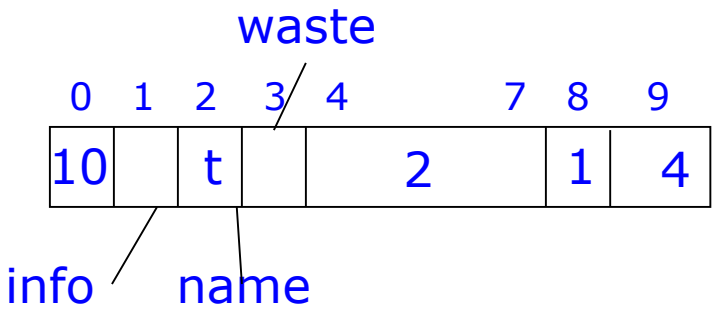
- Comparison:
  - The former strategy uses less space but is less efficient in locating fields: to locate a variable-length field beyond the first, we must examine all previous variable-length fields.
  - With the latter, fields can be located faster but pointers have to be stored with the record.
- Example: assume the square of a number is represented as a character string instead of an integer (variable-length field).

  A count for the length of this field and the total length of the record must be used according to the first strategy.

The Physical Data Model



Record for number 13



Record for number 2

The value of byte 0 is always 9 more than the value of byte 8. Hence, either one can be dispensed with (but not both).

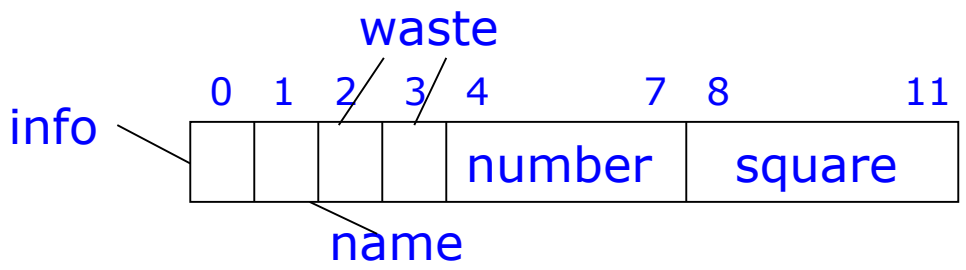
If there were additional fields following the square of the number, then byte 8 would have to be read before that field could be found.

The Physical Data Model

- Block formats
  - Records within blocks need to be located efficiently as well.
  - Just as records require additional space for formatting information, so do blocks of records. E.g., blocks often have pointers in fixed positions to link blocks in lists of blocks.
  - The formatting of blocks must take into account the alignment requirements of record fields.
    - E.g., if integers within the records are required to start at an address divisible by 4, then we require that the offsets of integers within the records be divisible by 4 and that records begin at some offset that is also divisible by 4. Normally blocks begin at an address that is a multiple of a power of 2.

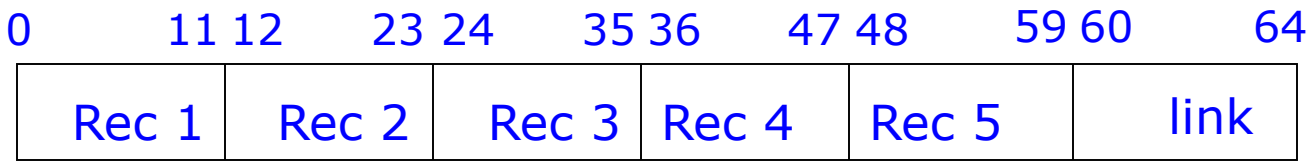
The Physical Data Model

- If a block contains fixed-length records, then the block has to be partitioned in as many areas, each holding one record, as will fit in a block.
- Space must be reserved for special fields in a known place in each block.
- Example: Assume blocks of length 64. We wish to store records of fixed length representing numbers.



The Physical Data Model

- Moreover, blocks must have a pointer of 4 bytes to be used as a link to another block.



- Every record contains a used/unused bit. To find an empty area in which to insert a record, we must visit each record area in turn.
- Alternatively, all used/unused bits can be grouped in one or more bytes at the beginning of the block.
- For the block shown above, only byte 0 is needed for storing the used/unused bits. Then the block can only contain 4 records because of the alignment requirements. (record length cannot be reduced below 12. Records stored in positions, 4-15,16-27,28-39,40-51. Bytes 60-63 for link. Bytes 52-59 are waste space.)

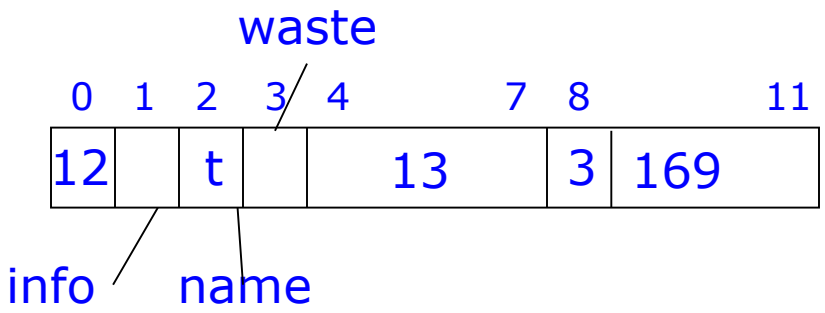
The Physical Data Model

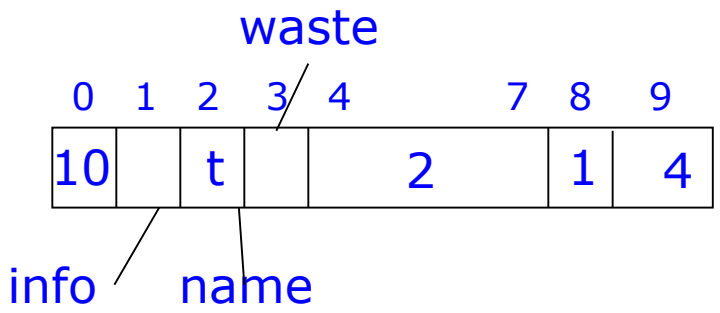
- Blocks with variable length records
  - Locating variable-length records:
    - Assume that the first record starts at byte 0. The length of the record is found there. The beginning of the second record is at the next multiple of 4 (after the number indicating the length of the first record). The third record is found at the next multiple of 4 following the second record, etc.
  - Alternatively, a directory can be placed at the beginning of the block, consisting of an array of pointers to the records in the block. These pointers are actually offsets from the beginning of the block to the location where the particular record begins.

The Physical Data Model

- Directory Representation
  1. Precede the directory by a byte denoting the number of pointers in the directory
  2. Use a fixed number of fields at the beginning of the block for pointers to records. If there are fewer records in the block than fields for pointers, these fields will contain 0.
  3. Use a variable number of fields for pointers to records, with the last such field containing 0.
- Example: Suppose we want to store variable-length records of the following format:

The Physical Data Model





Block format: