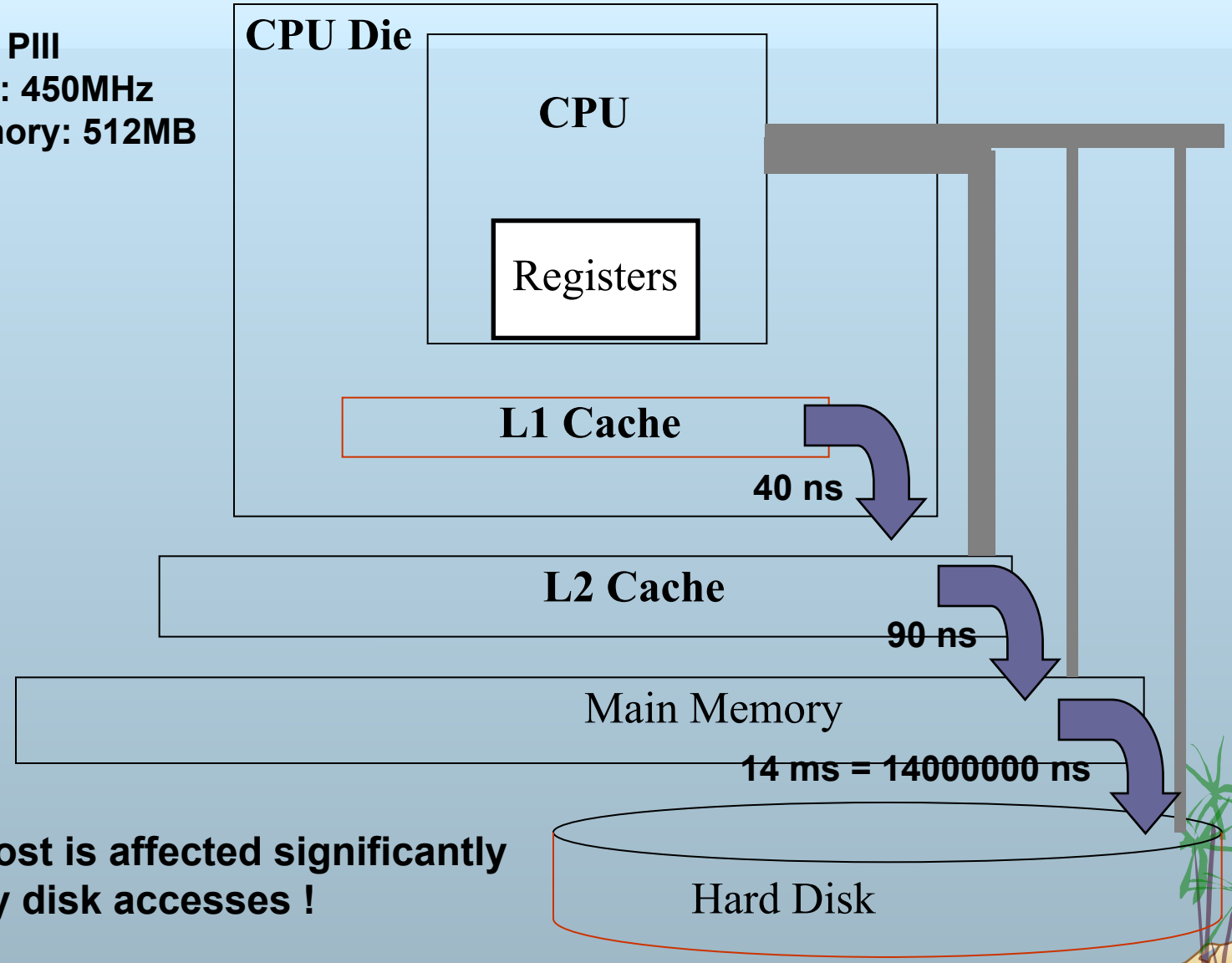
Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

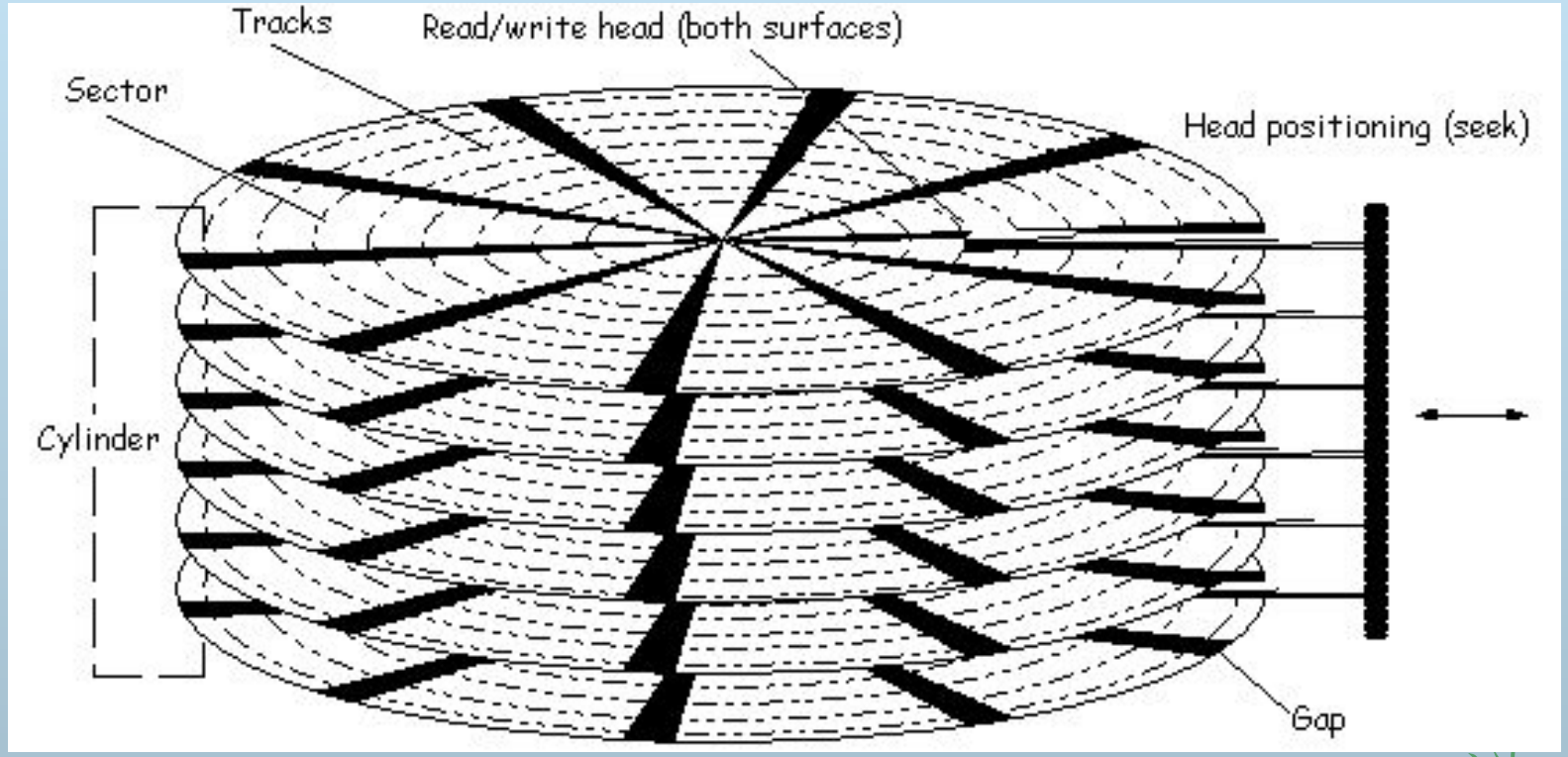
The Memory Hierarchy

Intel PIII
CPU: 450MHz
Memory: 512MB



Cost is affected significantly by disk accesses !

The Hard (Magnetic) disk



The Hard (Magnetic) disk

A block is a contiguous sequence of sectors from a single track of one platter. Block sizes range from 512 bytes to several kilobytes. Data are transferred between disk and main memory in units of blocks.

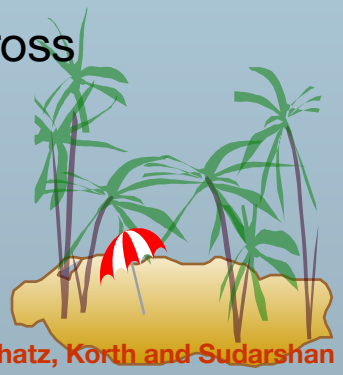
Reference: 11.2.3 Optimization of Disk-Block Access (Silberschatz–Korth–Sudarshan: Database System Concepts, Fourth Edition)

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".



Index Evaluation Metrics

- **Access types** supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- **Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

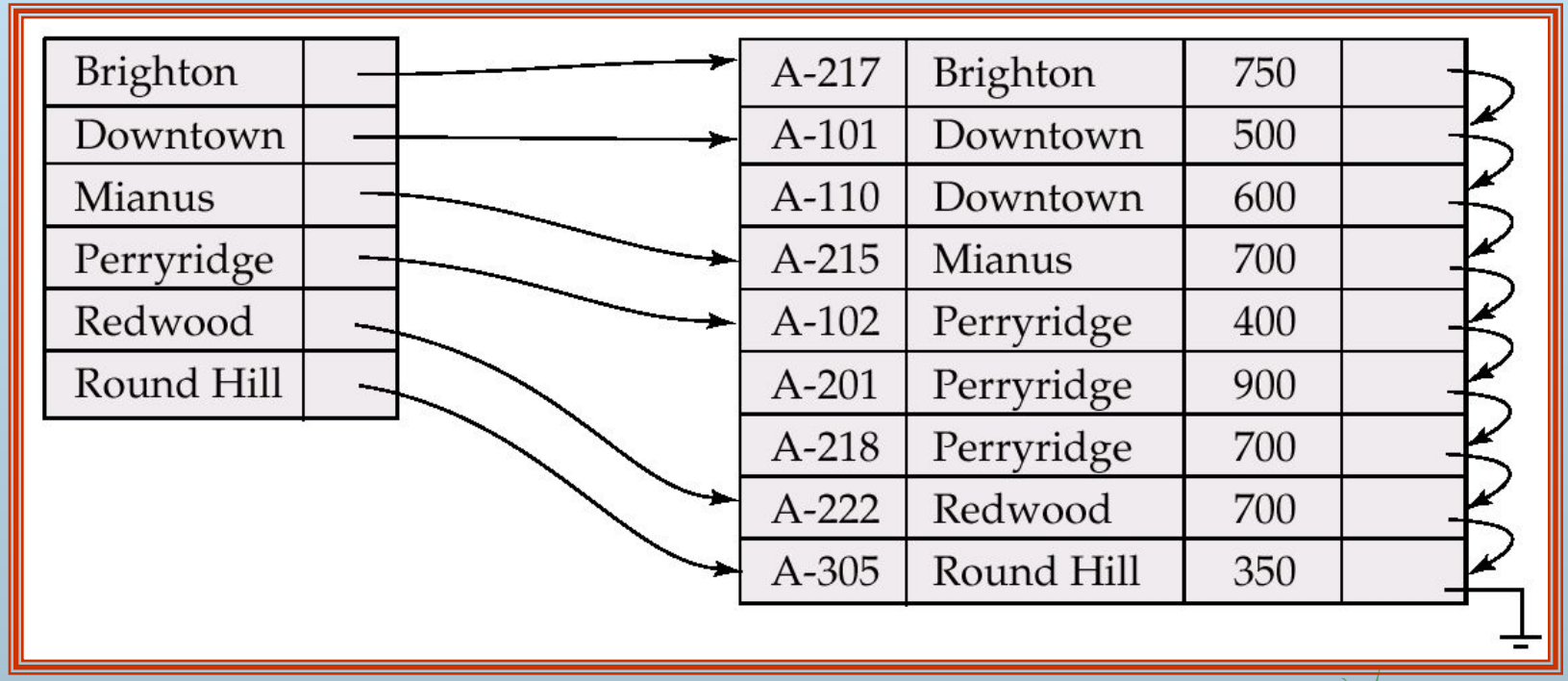
Ordered Indices

Indexing techniques evaluated on basis of:

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index.
- Index-sequential file: ordered sequential file with a primary index.

Dense Index Files

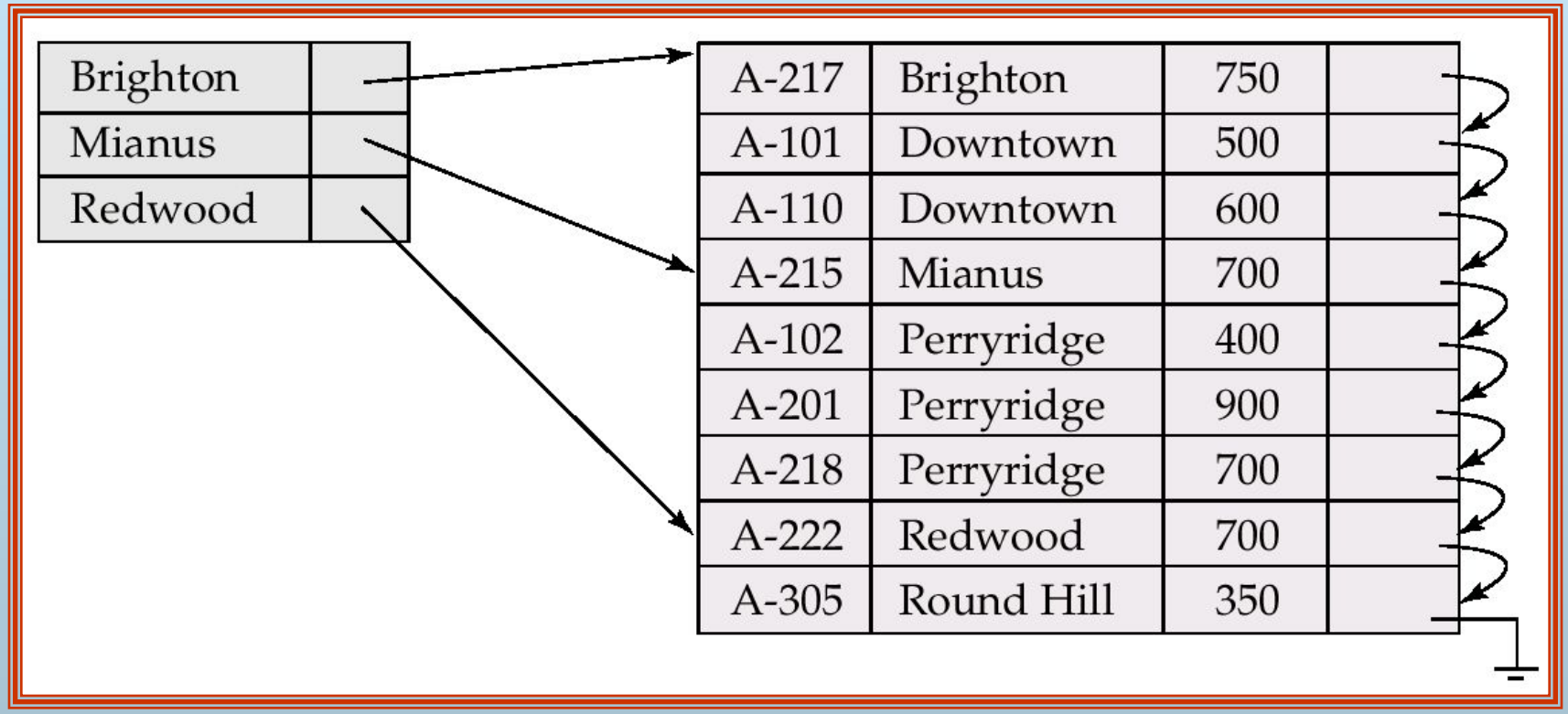
- Dense index — Index record appears for every search-key value in the file.



Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

Example of Sparse Index Files



Trade-off between Dense and Sparse primary index

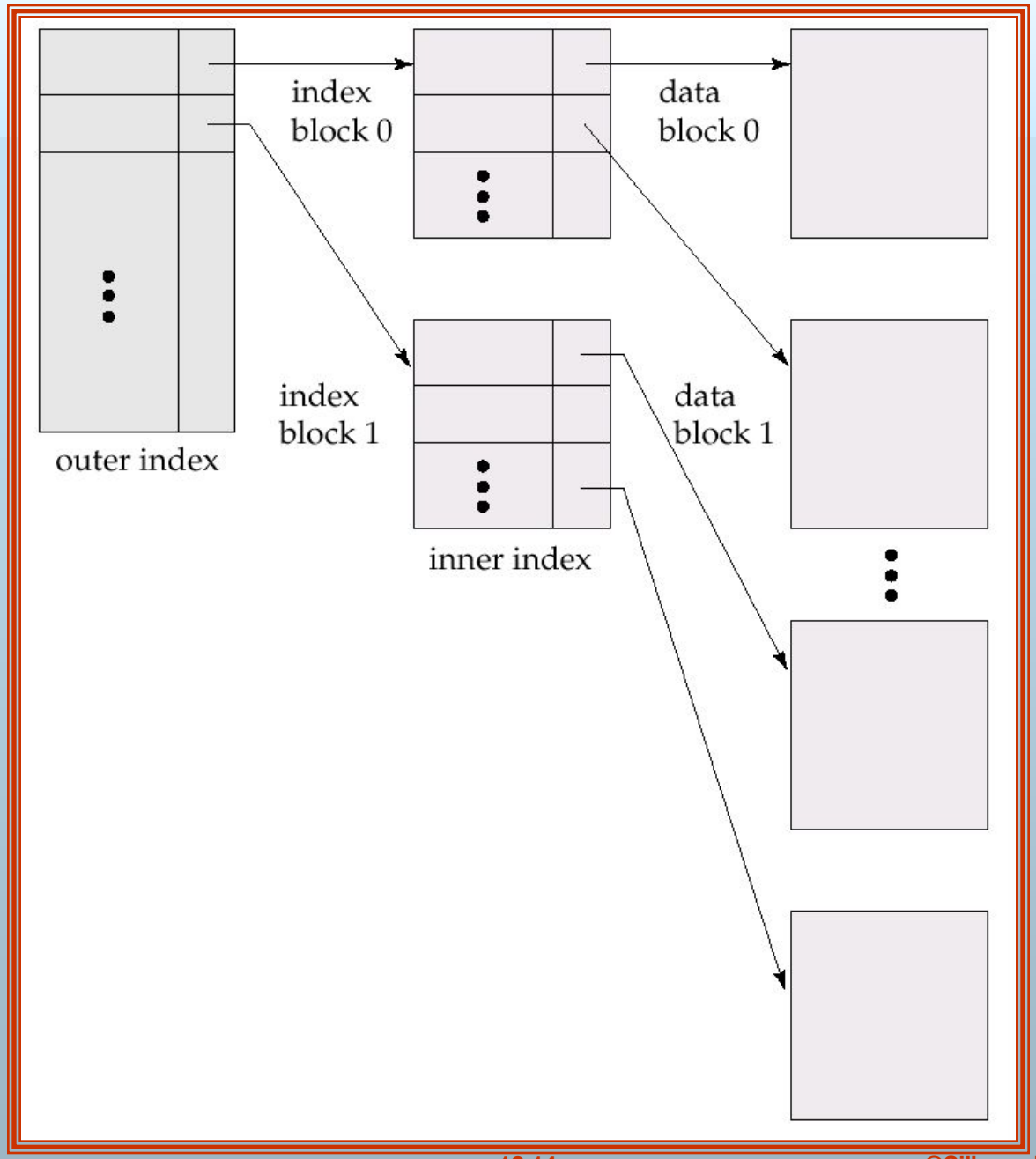
As we have seen, it is generally faster to locate a record if we have a dense index rather than a sparse index. However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.
There is a trade-off that the system designer must make between access time and space overhead. Although the decision regarding this trade-off depends on the specific application, a good compromise is to have a sparse index with one index entry per block.

The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once we have brought in the block, the time to scan the entire block is negligible.

Multilevel Index

- If index does not fit in main memory, access becomes expensive.
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a **sparse index** of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion

- **Deletion.** To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:
  - Dense indices:
    1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index record from the index.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system deletes the pointer to the deleted record from the index record.
       b. Otherwise, the index record stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index record to point to the next record.

Index Update: Deletion

- Sparse indices:
  1. If the index does not contain an index record with the search-key value of the deleted record, nothing needs to be done to the index.
  2. Otherwise the system takes the following actions:
     a. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
     b. Otherwise, if the index record for the search-key value points to the record being deleted, the system updates the index record to point to the next record with the same search-key value.

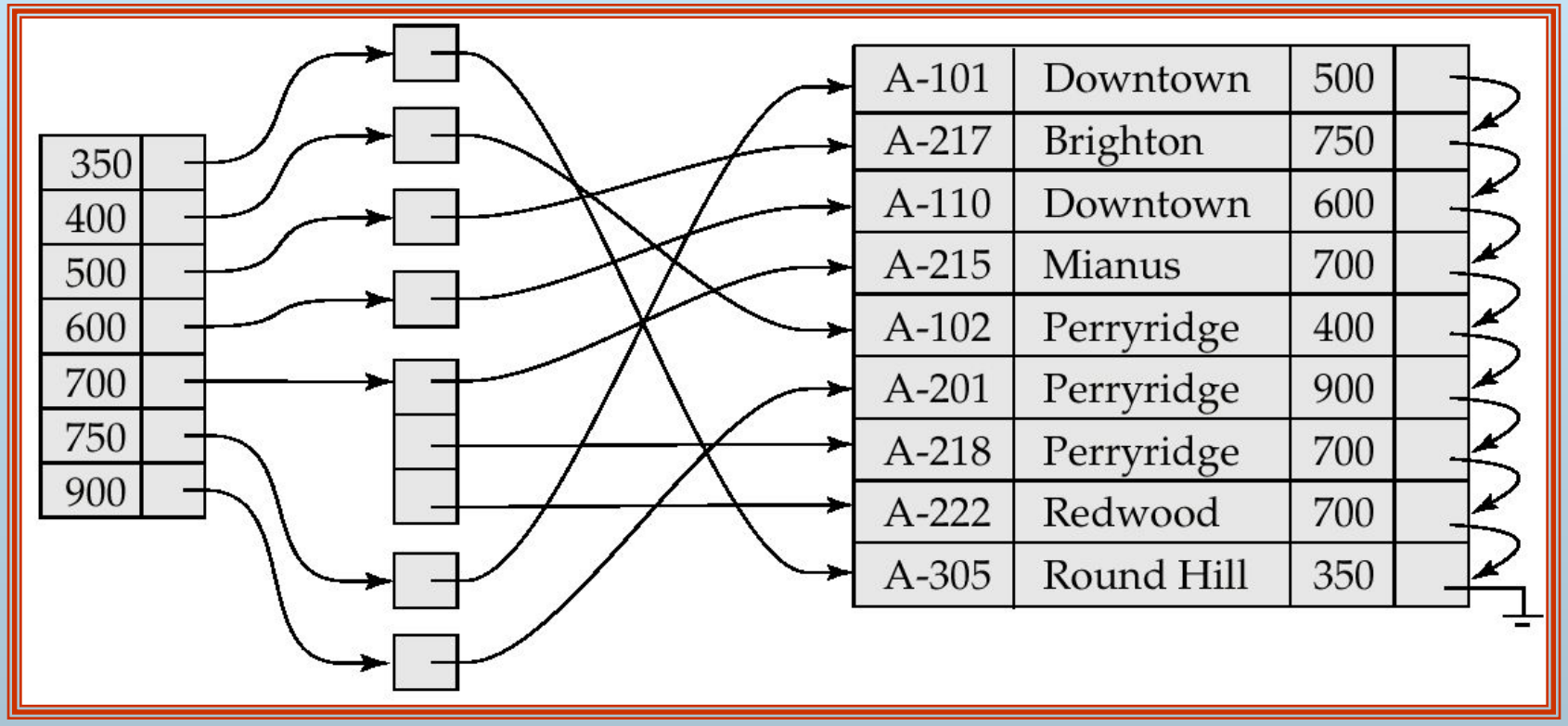
Index Update: Insertion

- **Insertion**. First, the system performs a lookup using the search-key value that appears in the record to be inserted. Again, the actions the system takes next depend on whether the index is dense or sparse:
  - ☐ Dense indices:
    1. If the search-key value does not appear in the index, the system inserts an index record with the search-key value in the index at the appropriate position.
    2. Otherwise the following actions are taken:
       a. If the index record stores pointers to all records with the same search-key value, the system adds a pointer to the new record to the index record.
       b. Otherwise, the index record stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.
  - ☐ Sparse indices: We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary Index on *balance* field of *account*



Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - each record access may fetch a new block from disk

A sequential file

A sequential file is designed for efficient processing of records in sorted order based on some search-key. A search key is any attribute or set of attributes; it need not be the primary key, or even a superkey. To permit fast retrieval of records in search-key order, we chain together records by pointers. The pointer in each record points to the next record in search-key order.

A sequential file

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Figure 11.15 Sequential file for *account* records.

A sequential file

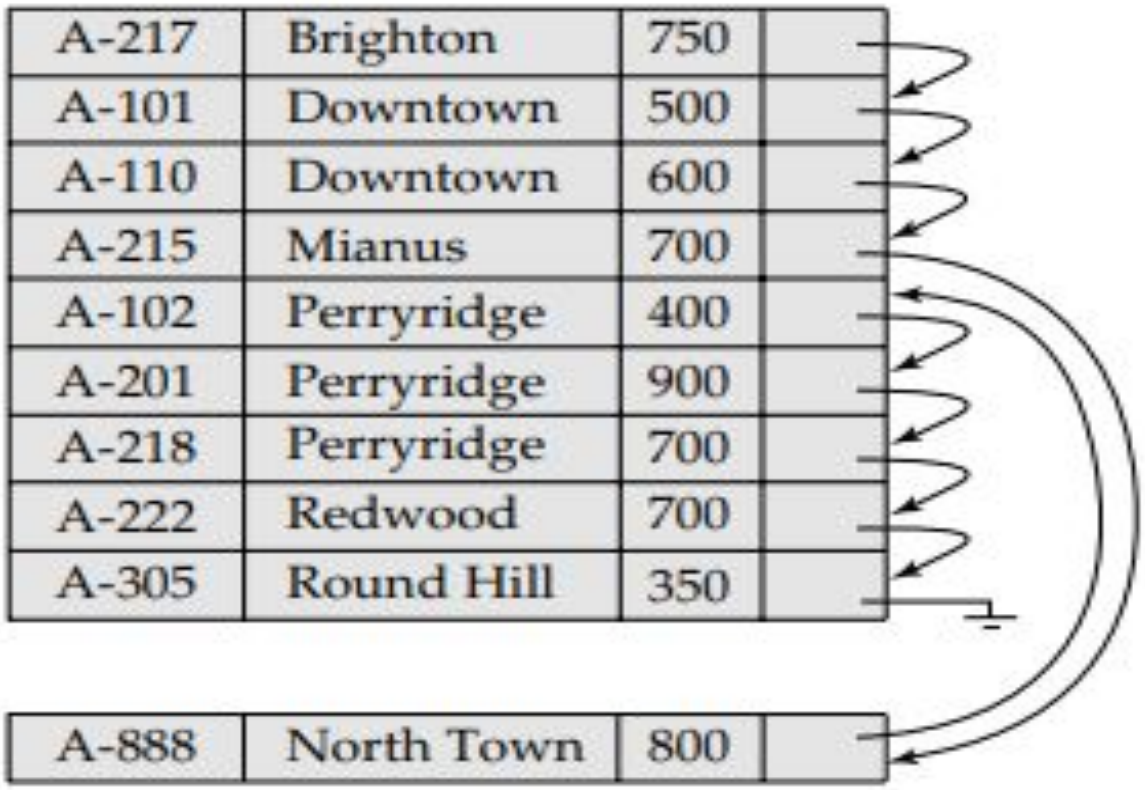


Figure 11.16 Sequential file after an insertion.

A sequential file

For insertion, we apply the following rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.

2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an overflow block. In either case, adjust the pointers so as to chain together the records in search-key order.

B^+-Tree Index Files

B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files: performance degrades as file grows, since many overflow blocks get created. Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files: automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. Reorganization of entire file is not required to maintain performance.
- Disadvantage of B^+-trees: extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages, and they are used extensively.

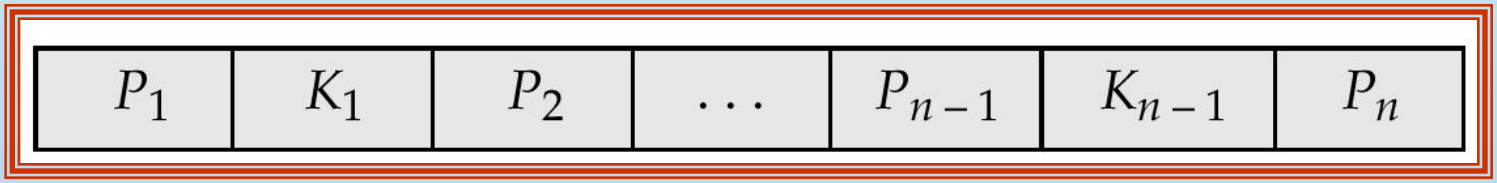
B^+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $[n/2]$ and n children.
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-Tree Node Structure

- Typical node



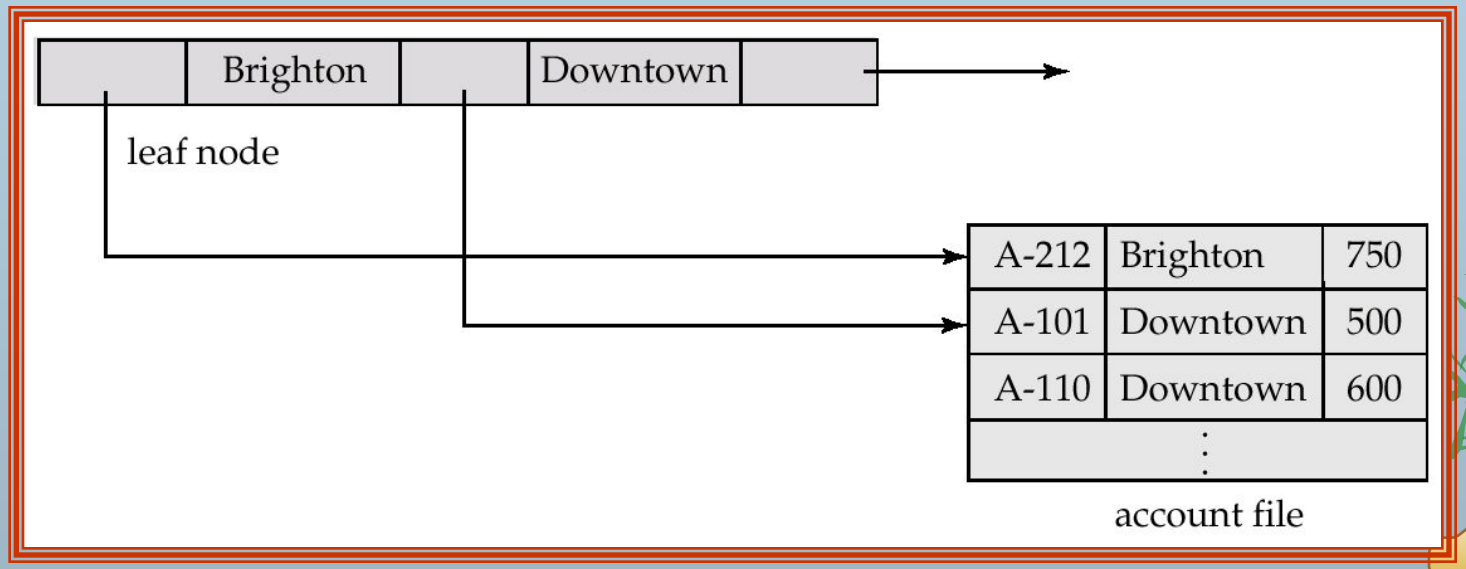
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Leaf Nodes in B+-Trees

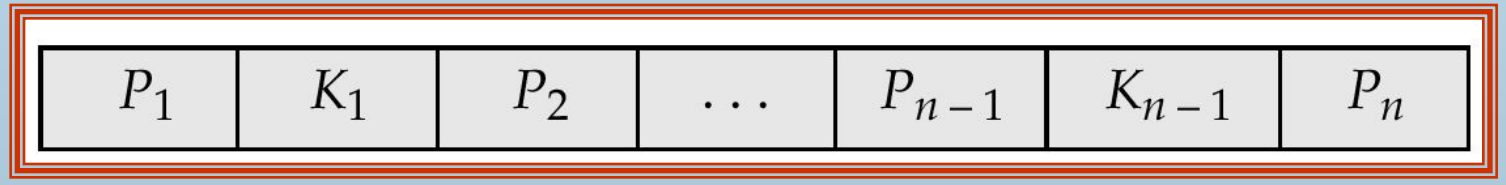
Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order

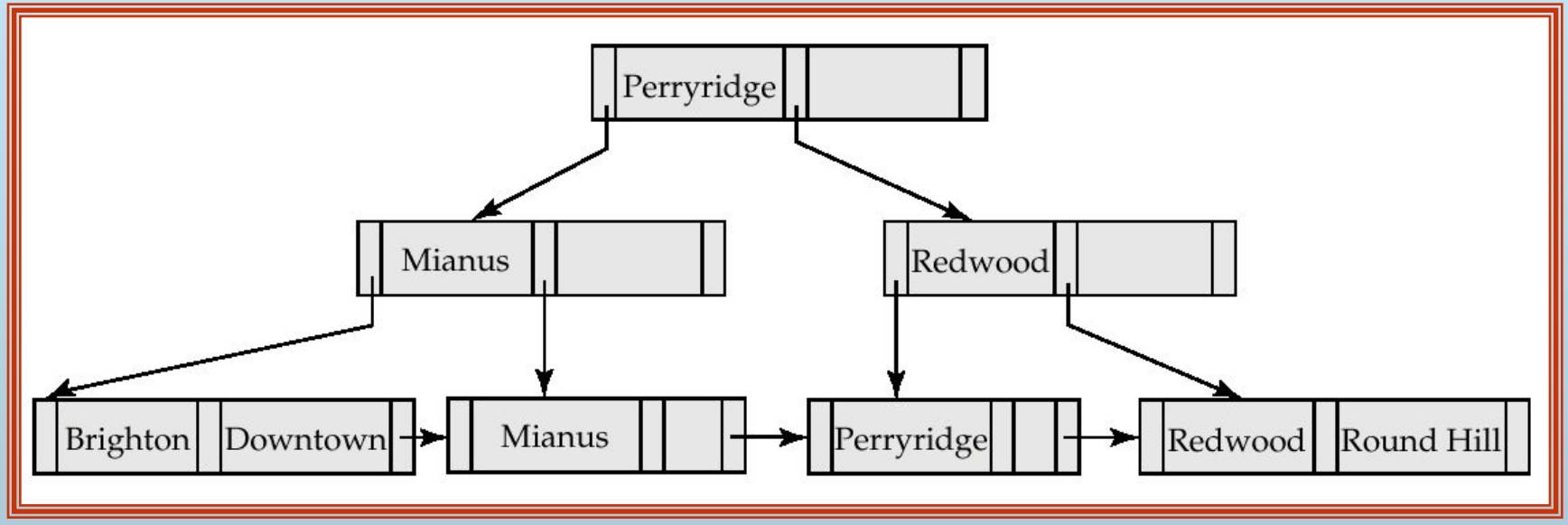


Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \le i \le n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_{m-1}

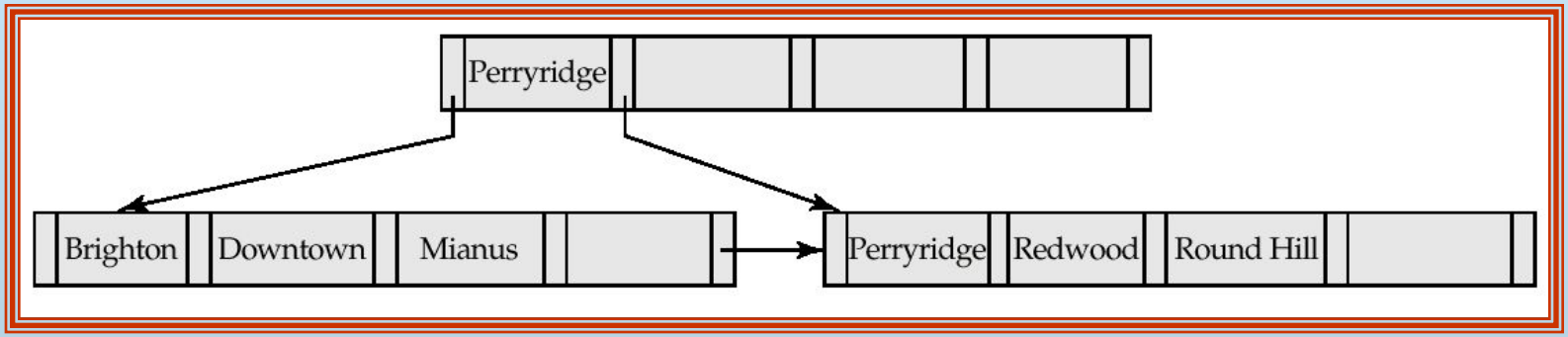


Example of a B+-tree



B+-tree for *account* file (*n* = 3)

Example of B+-tree



B+-tree for *account* file (*n* = 5)

- Non-leaf nodes other than root must have between 3 and 5 children (⌈(*n*/2⌉ and *n* with *n* =5).
- Root must have at least 2 children.

Observations about B^+-trees

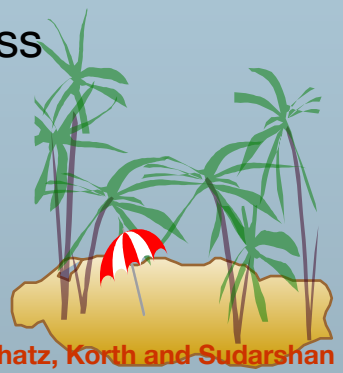
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Queries on B$^+$-Trees

- Find all records with a search-key value of *k.*
  1. Start with the root node
     1. Examine the node for the smallest search-key value > *k.*
     2. If such a value exists, assume it is K_j. Then follow P_i to the child node
     3. Otherwise $k \geq K_{m-1}$, where there are *m* pointers in the node. Then follow P_m to the child node.
  2. If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  3. Eventually reach a leaf node. If for some *i*, key K_i = k follow pointer P_i to the desired record or bucket. Else no record with search-key value *k* exists.

Queries on B+-Trees (Cont.)

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes, and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$, at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary free with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds!



Why B+-Tree Index?

- B+-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B+-tree structure.

Why B+-Tree Index?

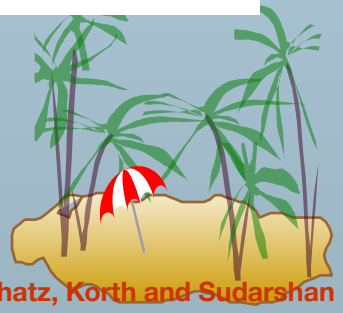
Thus, in processing a query, we traverse a path in the tree from the root to some leaf node. If there are K search-key values in the file, the path is no longer than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

In practice, only a few nodes need to be accessed, Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes. With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes, n is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, n is around 100. With $n = 100$, if we have 1 million search-key values in the file, a lookup requires only

$\lceil \log_{50}(1{,}000{,}000) \rceil = 4$ nodes to be accessed. Thus, at most four blocks need to be read from disk for the lookup. The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.

Why B$^+$-Tree Index?

An important difference between B$^+$-tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree. In a binary tree, each node is small, and has at most two pointers. In a B$^+$-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B$^+$-trees tend to be fat and short, unlike thin and tall binary trees. In a balanced binary tree, the path for a lookup can be of length $\lceil \log_2(K) \rceil$, where K is the number of search-key values. With $K = 1{,}000{,}000$ as in the previous example, a balanced binary tree requires around 20 node accesses. If each node were on a different disk block, 20 block reads would be required to process a lookup, in contrast to the four block reads for the B$^+$-tree.

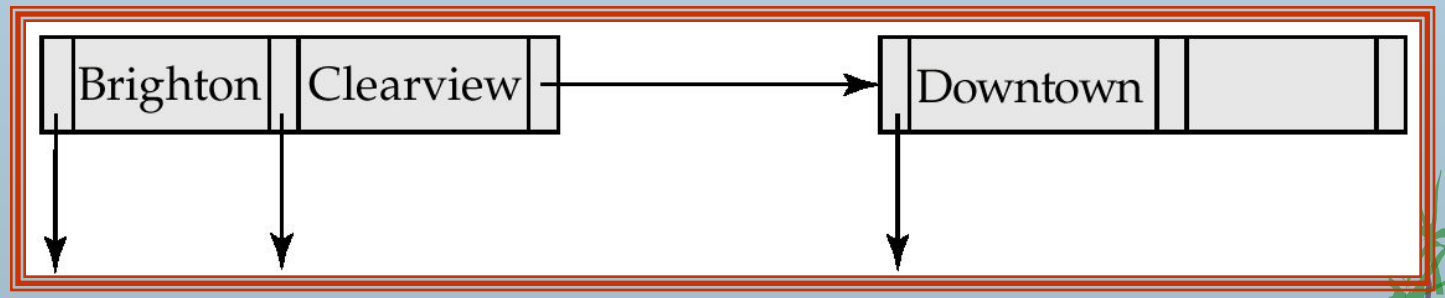


Updates on B+-Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- If the search-key value is not there, then add the record to the main file and create a bucket if necessary. Then:
  - If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  - Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

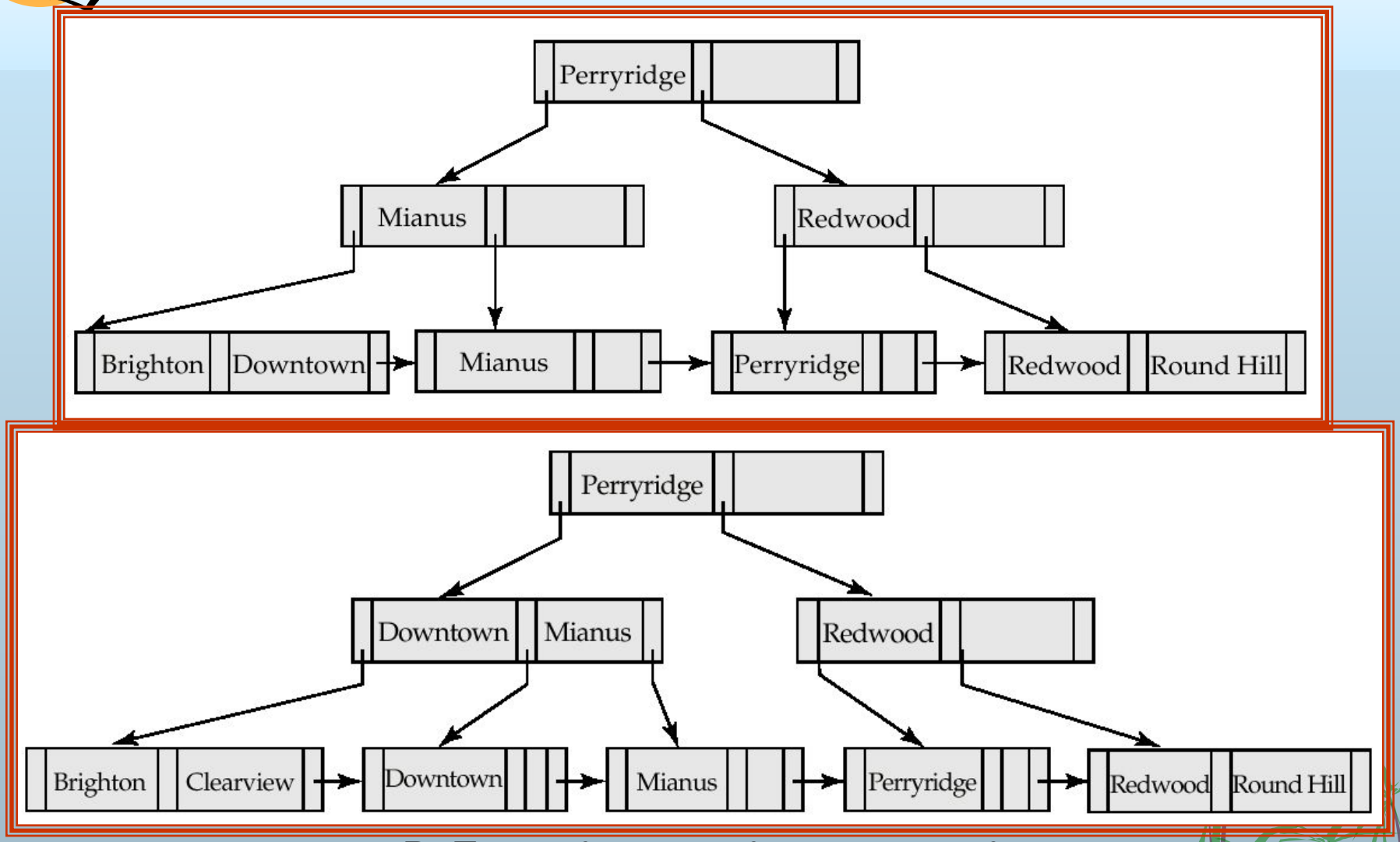
Updates on B+-Trees: Insertion (Cont.)

- Splitting a node:
  - take the *n*(search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split. If the parent is full, split it and propagate the split further up.
- The splitting of nodes proceeds upwards till a node that is not full is found. In the worst case the root node may be split increasing the height of the tree by 1.



Result of splitting node containing Brighton and Downtown on
inserting Clearview

Updates on B+-Trees: Insertion (Cont.)



B+-Tree before and after insertion of "Clearview"

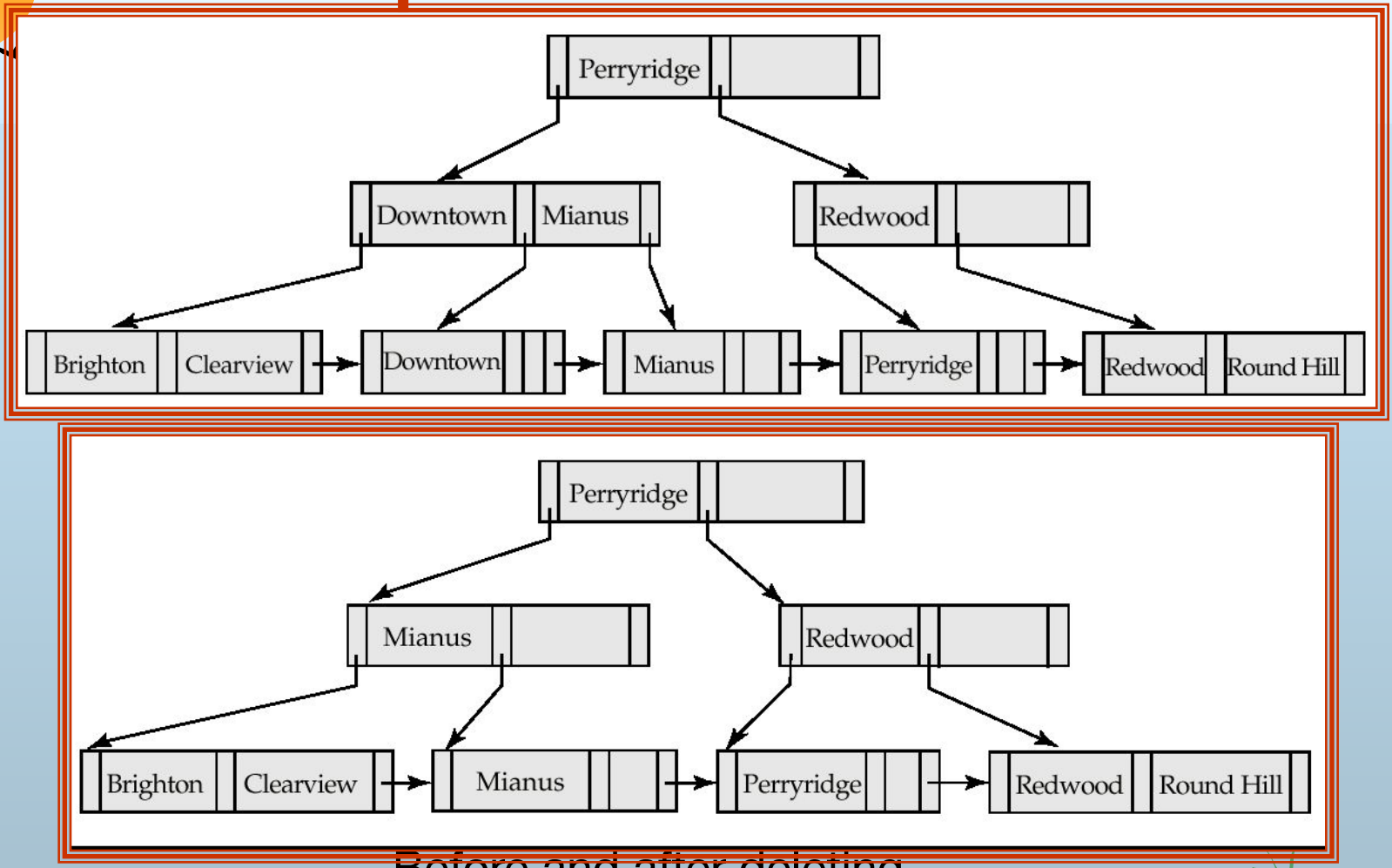
Updates on B$^+$-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

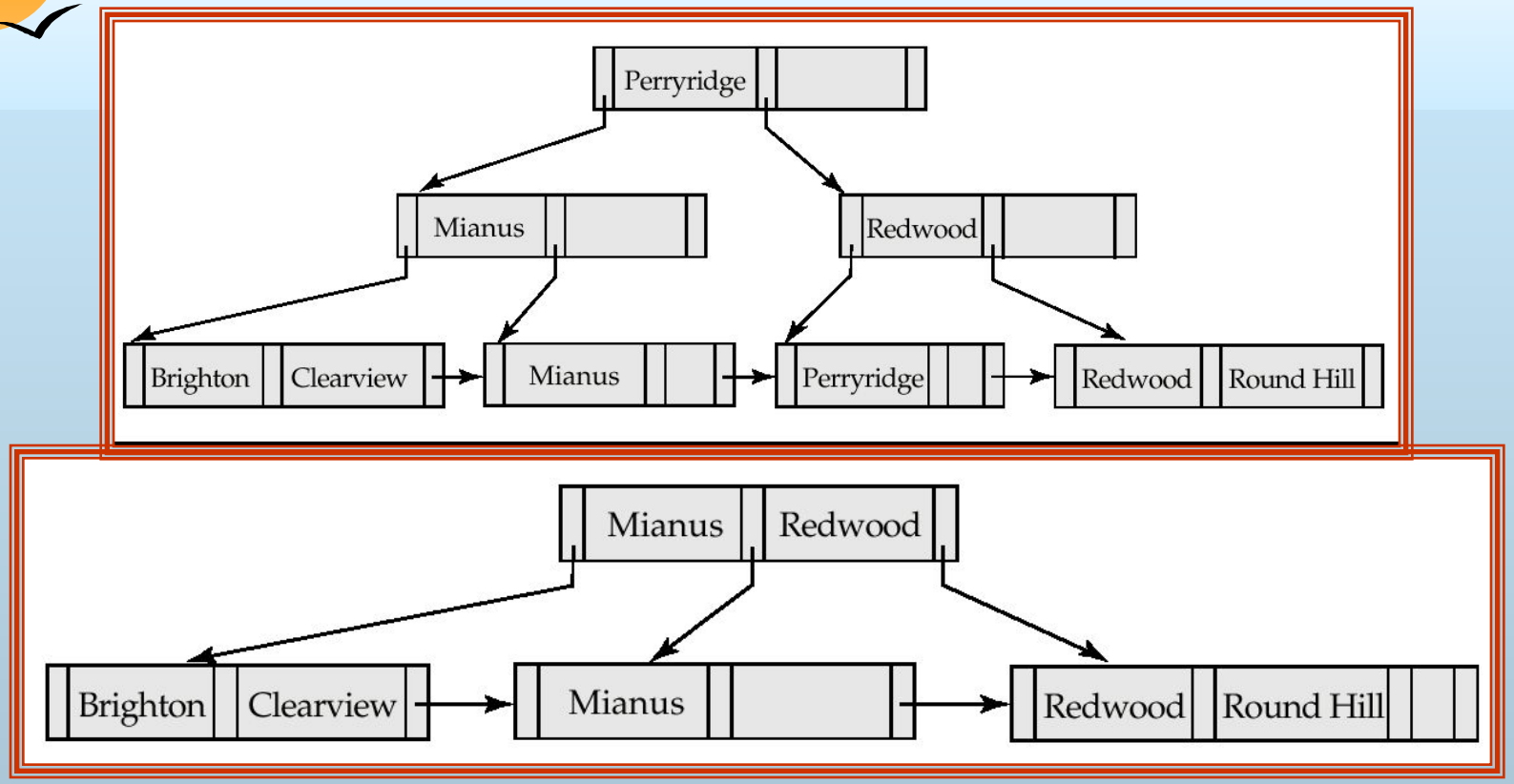
Examples of B+-Tree Deletion



Before and after deleting "Downtown"

- The removal of the leaf node containing "Downtown" did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node's parent.

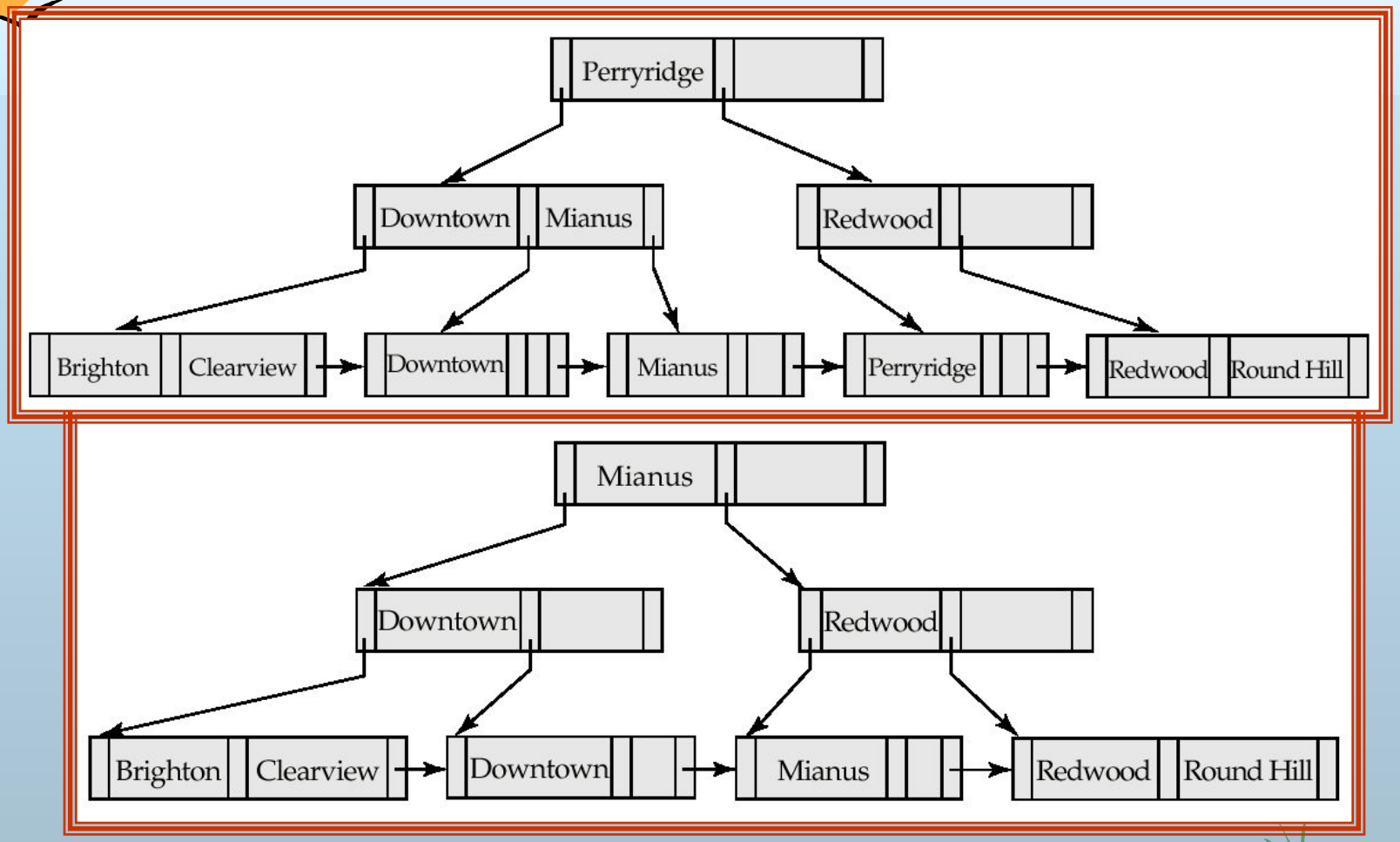
Examples of B+-Tree Deletion (Cont.)



Deletion of "Perryridge" from result of previous example

- Node with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling (and an entry was deleted from their parent)
- Root node then had only one child, and was deleted and its child became the new root node

Example of B+-tree Deletion (Cont.)



Before and after deletion of "Perryridge" from earlier example

- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

Problem of Index Sequential File

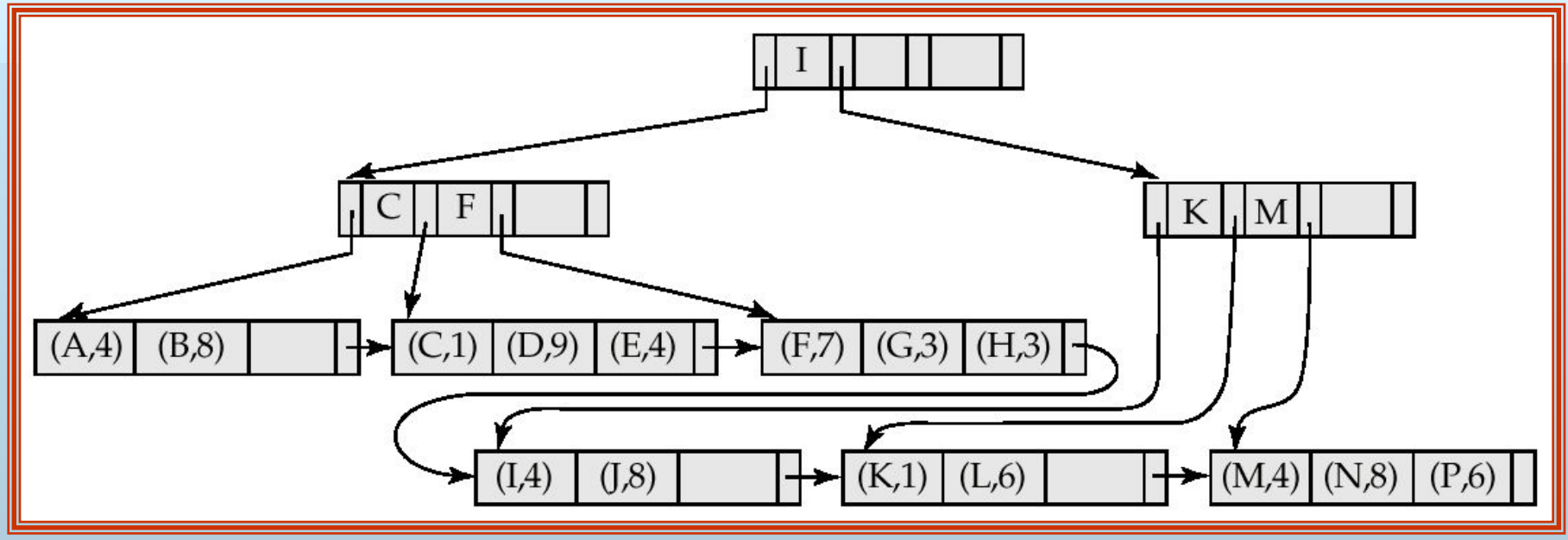
The main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index records and actual records become out of order, and are stored in overflow blocks.

We solve the degradation of index lookups by using B+-tree indices on the file. We solve the degradation problem for storing the actual records by using the leaf level of the B+-tree to organize the blocks containing the actual records.

B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices. Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Leaf nodes are still required to be at least half full.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)



Example of B+-tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

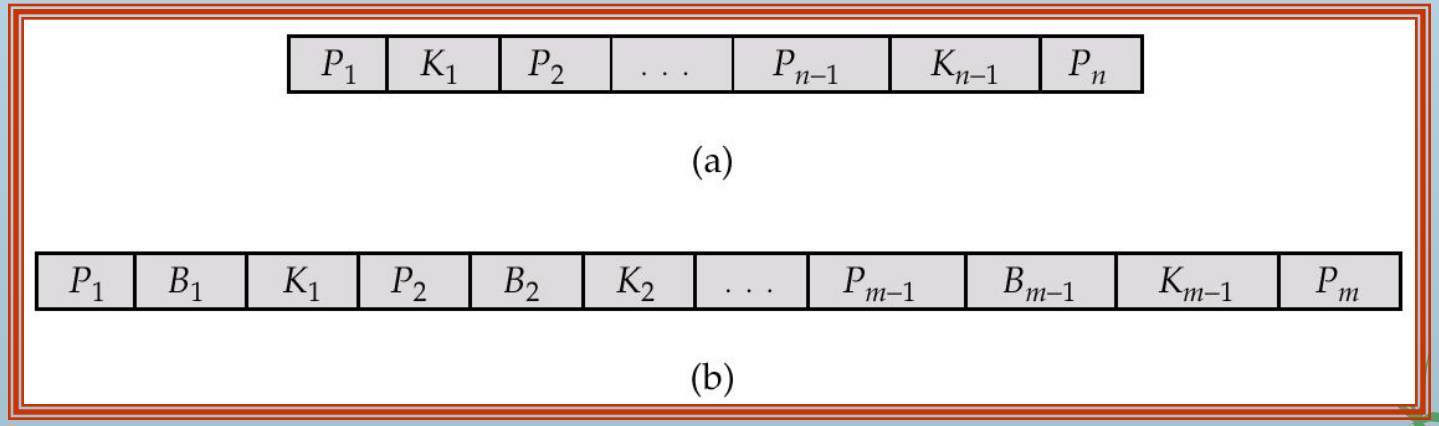
B+-Tree File Organization (Cont.)

When we use a B^+-tree for file organization, space utilization is particularly important, since the space occupied by the records is likely to be much more than the space occupied by keys and pointers. We can improve the utilization of space in a B^+-tree by involving more sibling nodes in redistribution during splits and merges. The technique is applicable to both leaf nodes and internal nodes, and works as follows.

During insertion, if a node is full the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry. If this attempt fails because the adjacent nodes are themselves full, the system splits the node, and splits the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node. Since the three nodes together contain one more record than can fit in two nodes, each node will be about two-thirds full. More precisely, each node will have at least $\lfloor 2n/3 \rfloor$ entries, where n is the maximum number of entries that the node can hold. ($\lfloor x \rfloor$ denotes the greatest integer that is less than or equal to x; that is, we drop the fractional part, if any.)

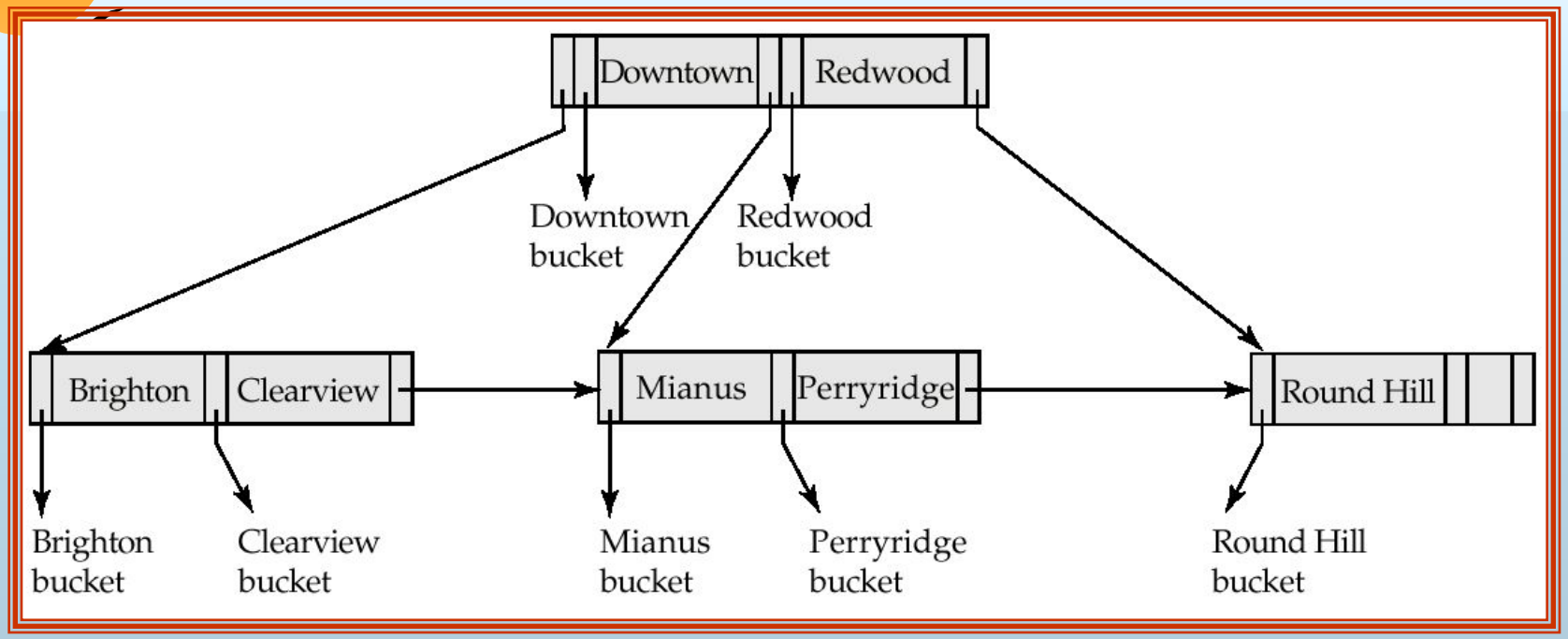
B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

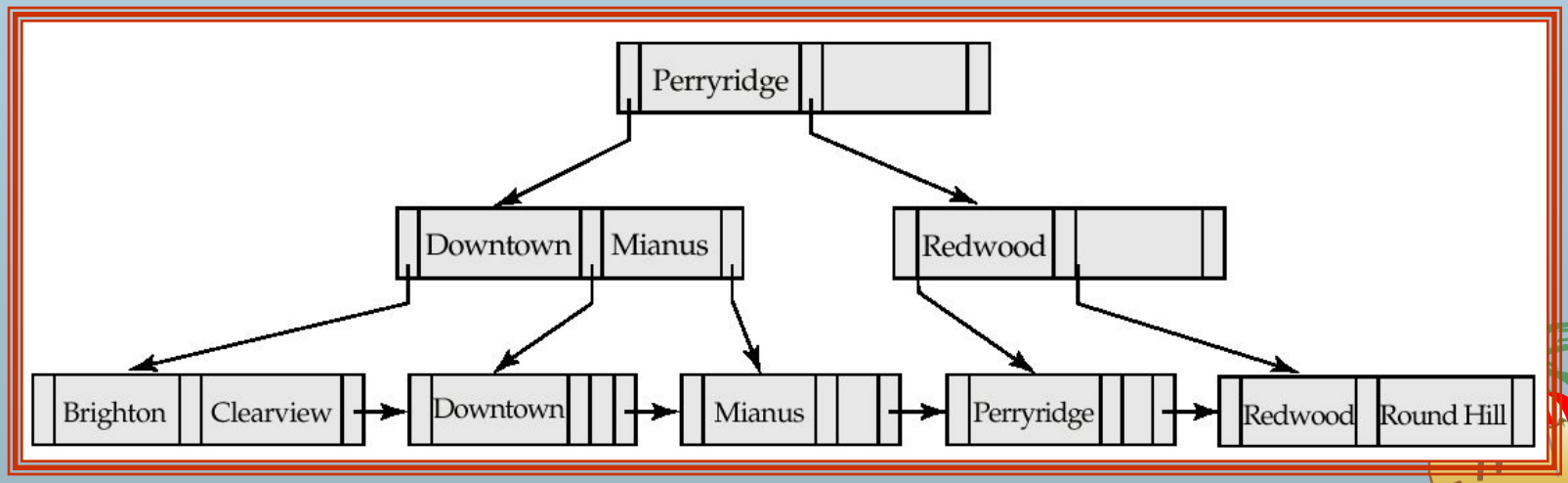


- Nonleaf node – pointers B_i are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

Static Hashing

- It can be used for two different purposes.
  - Hash File Organization

    The address of the disk block containing the actual record is obtained by using the hash function on a search key.

  - Hash Index Organization

    The search keys along with their pointers to actual records is organized in a hash file structure.

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block but could be larger or smaller than a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization (Cont.)

Hash file organization of *account* file, using *branch-name* as key

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i. e.g. A=>1*
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Example of Hash File Organization

- Hash file organization of *account* file, using *branch-name* as key

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of **search-key values** from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have nearly the same number of **records** assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

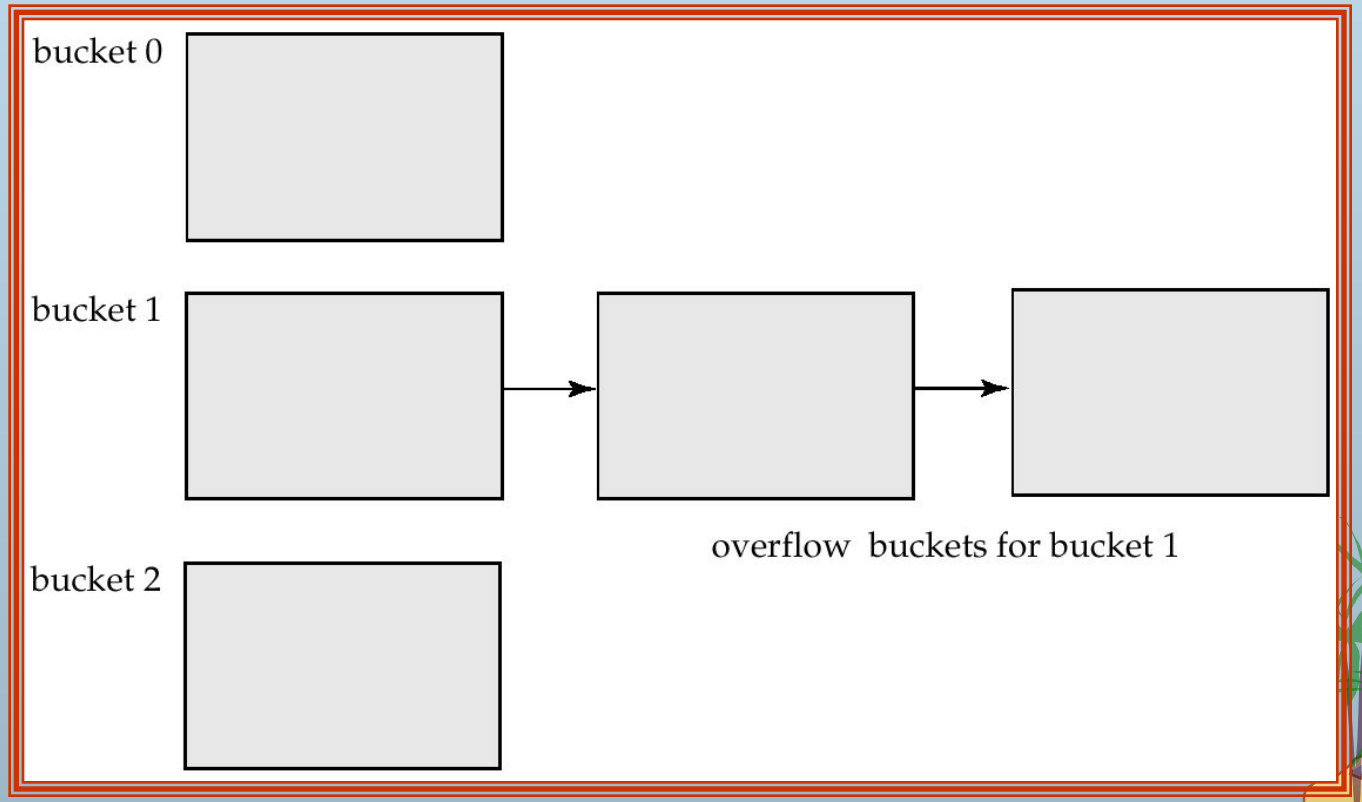
Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced by choosing the number of buckets to be (nr/fr)*(1+d) where d is around 0.2,

  it cannot be eliminated; it is handled by using *overflow buckets*.

Handling of Bucket Overflows (Cont.)

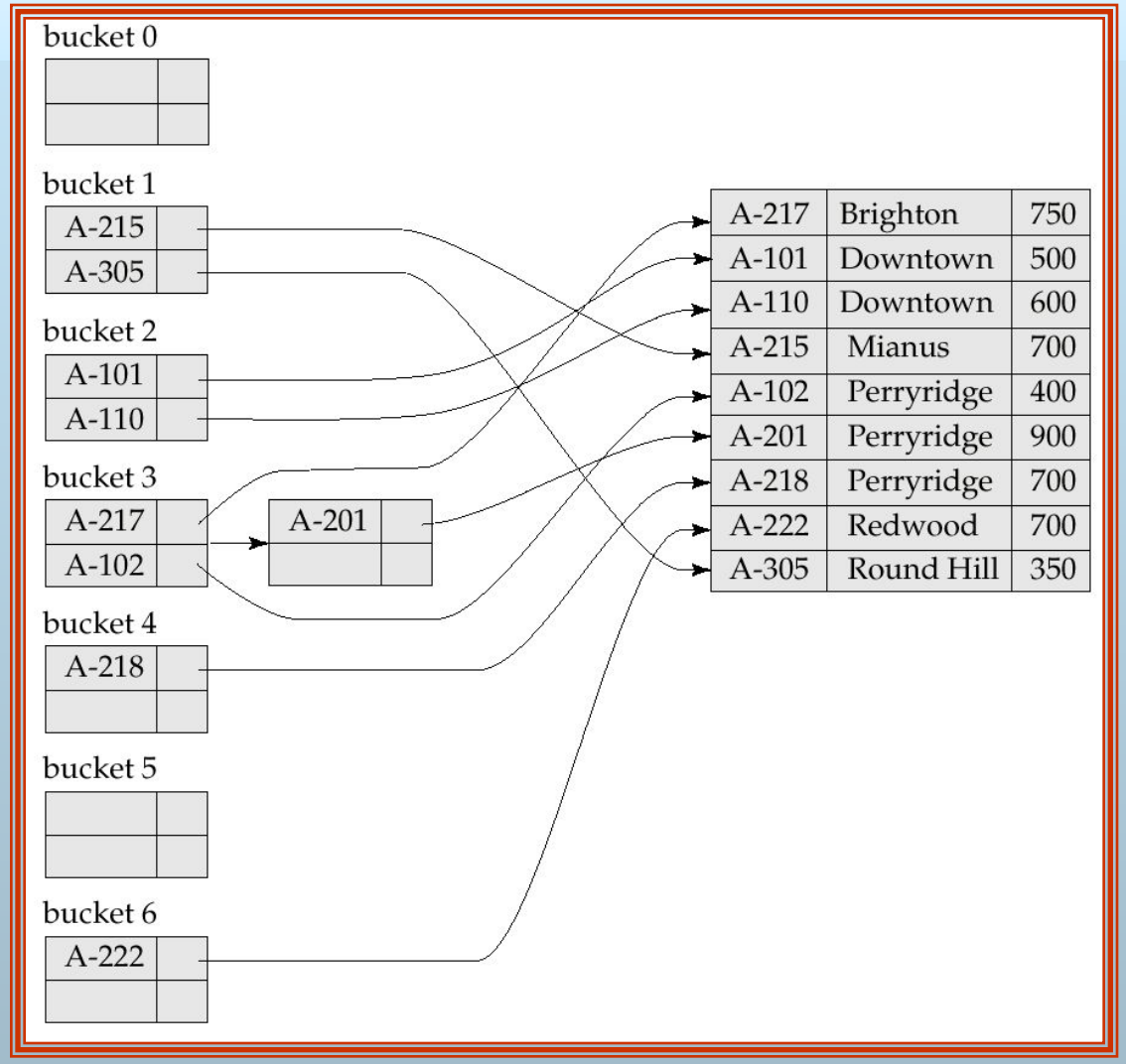
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.
  - An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary hash indices
  - if the file itself is organized using hashing, a separate hash index on it using the same search-key is unnecessary.
  - We use the term hash index to refer to both secondary hash indices and hash organized file .

Example of Hash Index



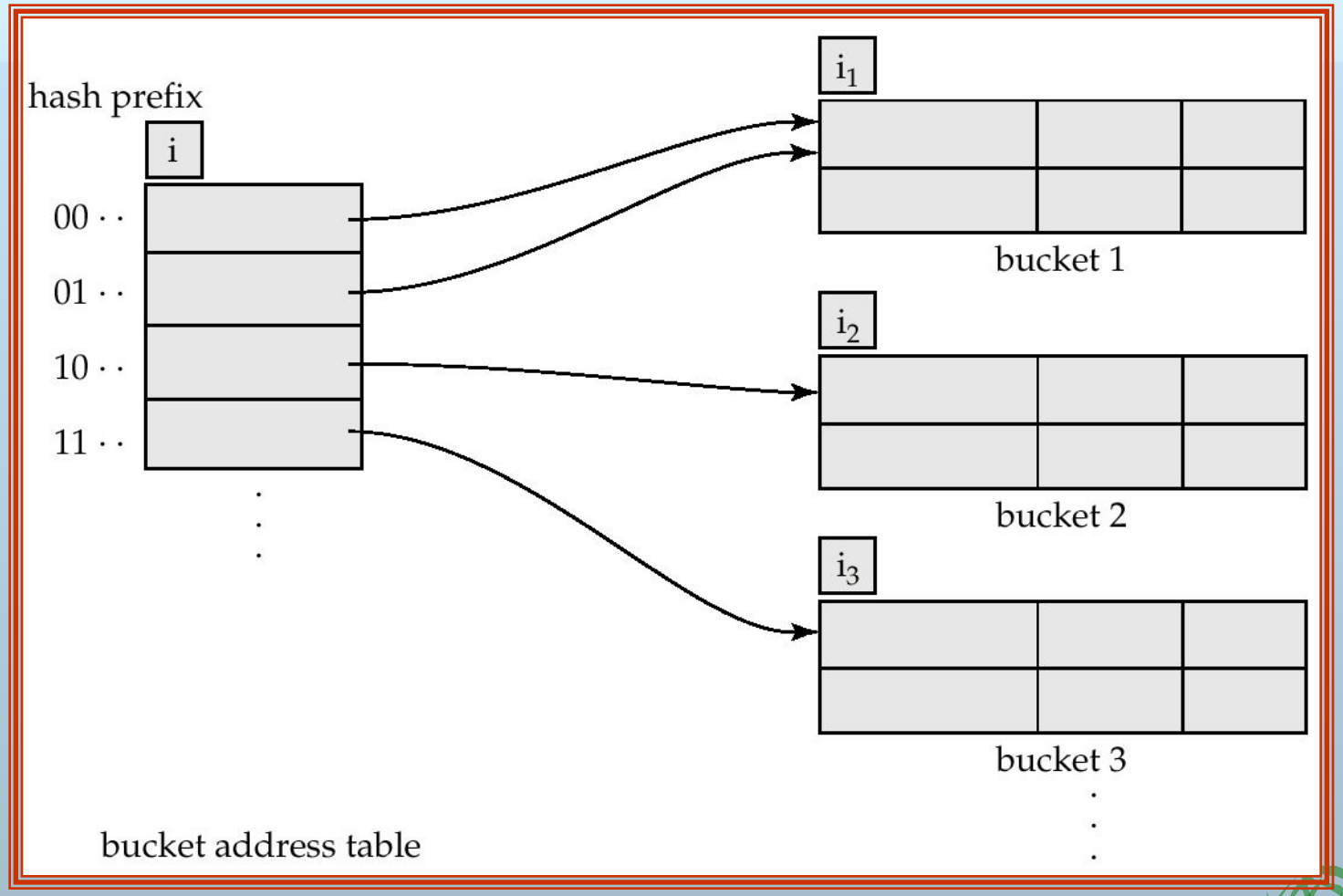
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \le i \le 32$.
  - Bucket address table size = 2^i. Initially $i = 0$
  - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket.
  - Thus, actual number of buckets is $< 2^i$
    - ★ The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket j stores a value i_j; all the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say j.
  - If there is room in the bucket j insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases (will see shortly)

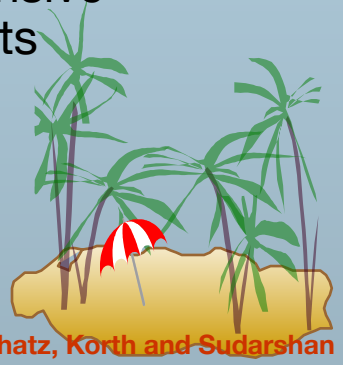
Updates in Extendable Hash Structure

To split a bucket j when inserting record with search-key value K_j:

- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set i_j and i_z to the old i_j -+ 1.
  - make the second half of the bucket address table entries pointing to j to point to z
  - remove and reinsert each record in bucket j.
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - increment i and double the size of the bucket address table.
  - replace each entry in the table by two entries that point to the same bucket.
  - recompute new bucket address table entry for K_j Now $i > i_j$ so use the first case above.

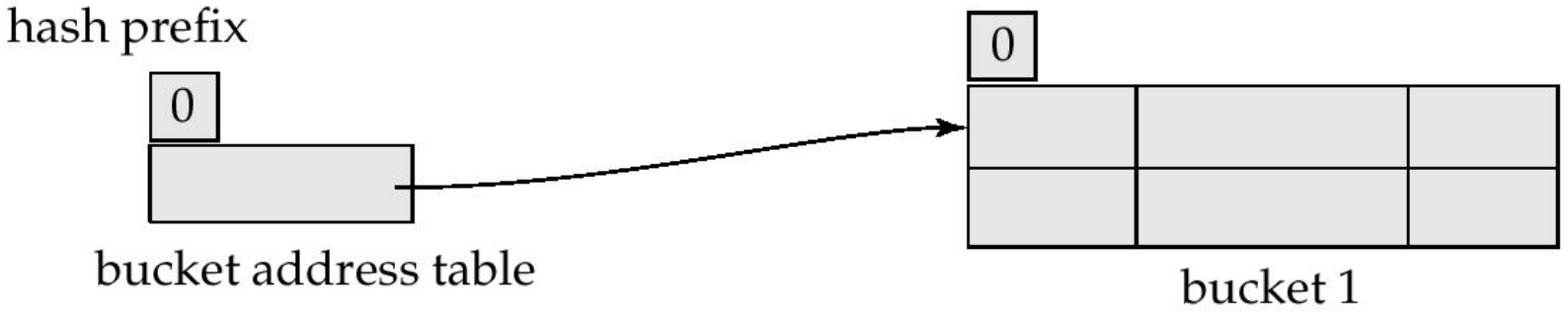
Updates in Extendable Hash Structure (Cont.)

- When inserting a value, if the bucket is full after several splits (that is, *i* reaches some limit *b*) create an overflow bucket instead of splitting bucket entry table further.
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - ★ Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table



Use of Extendable Hash Structure: Example

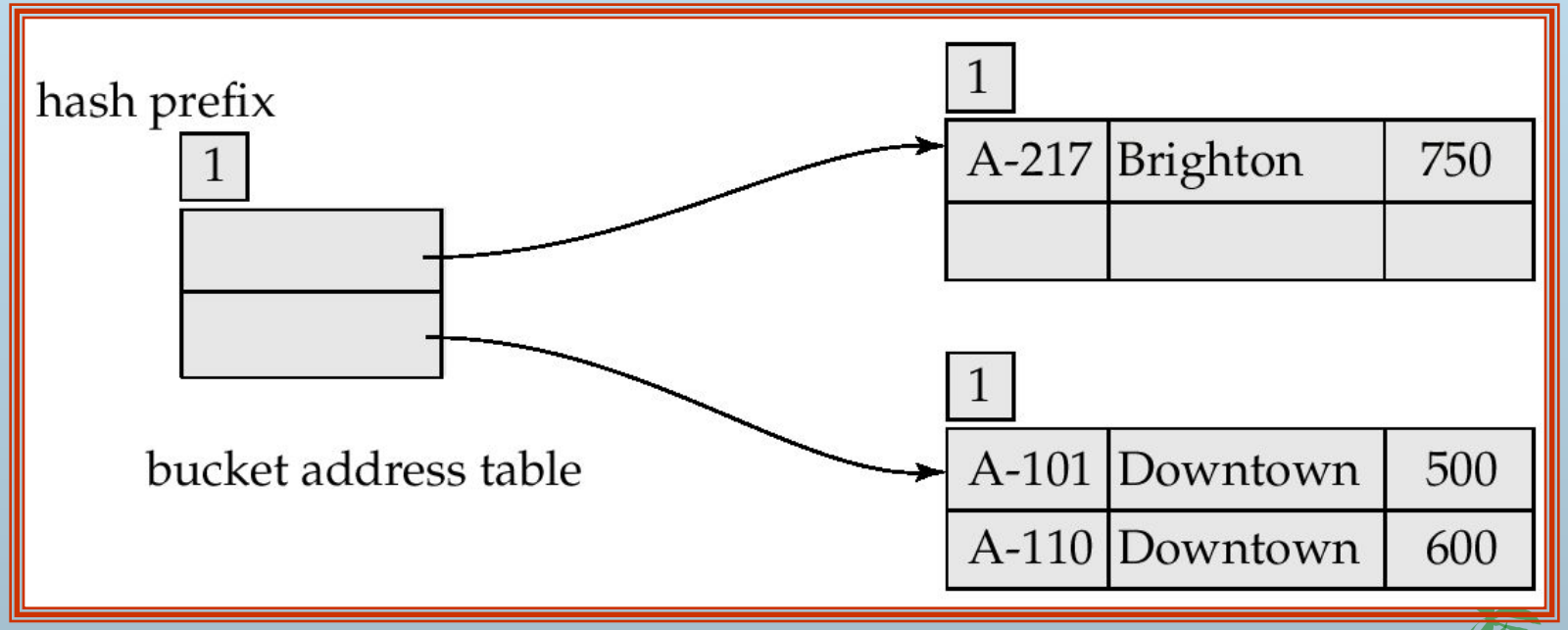
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

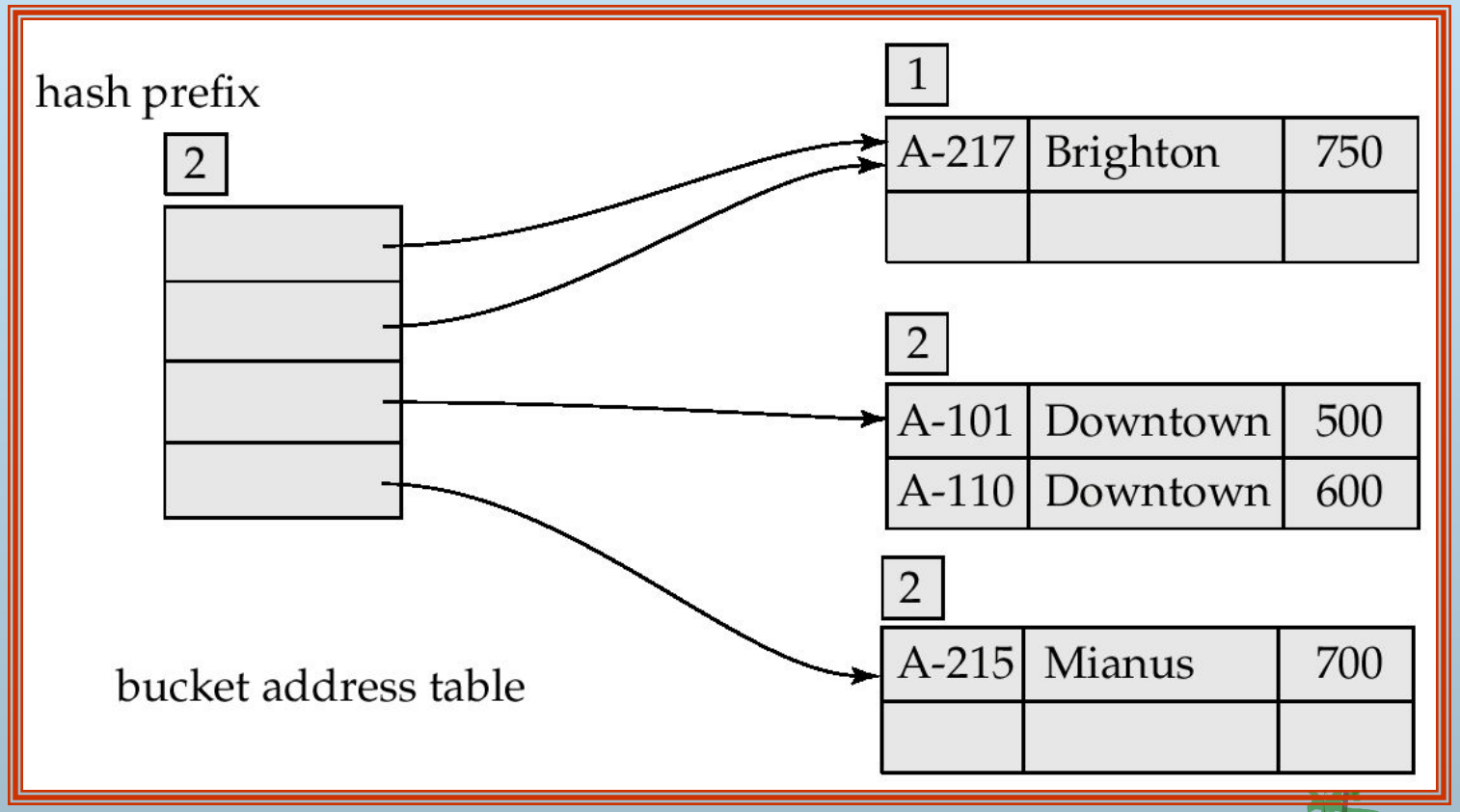
Example (Cont.)

- Hash structure after insertion of one Brighton and two Downtown records

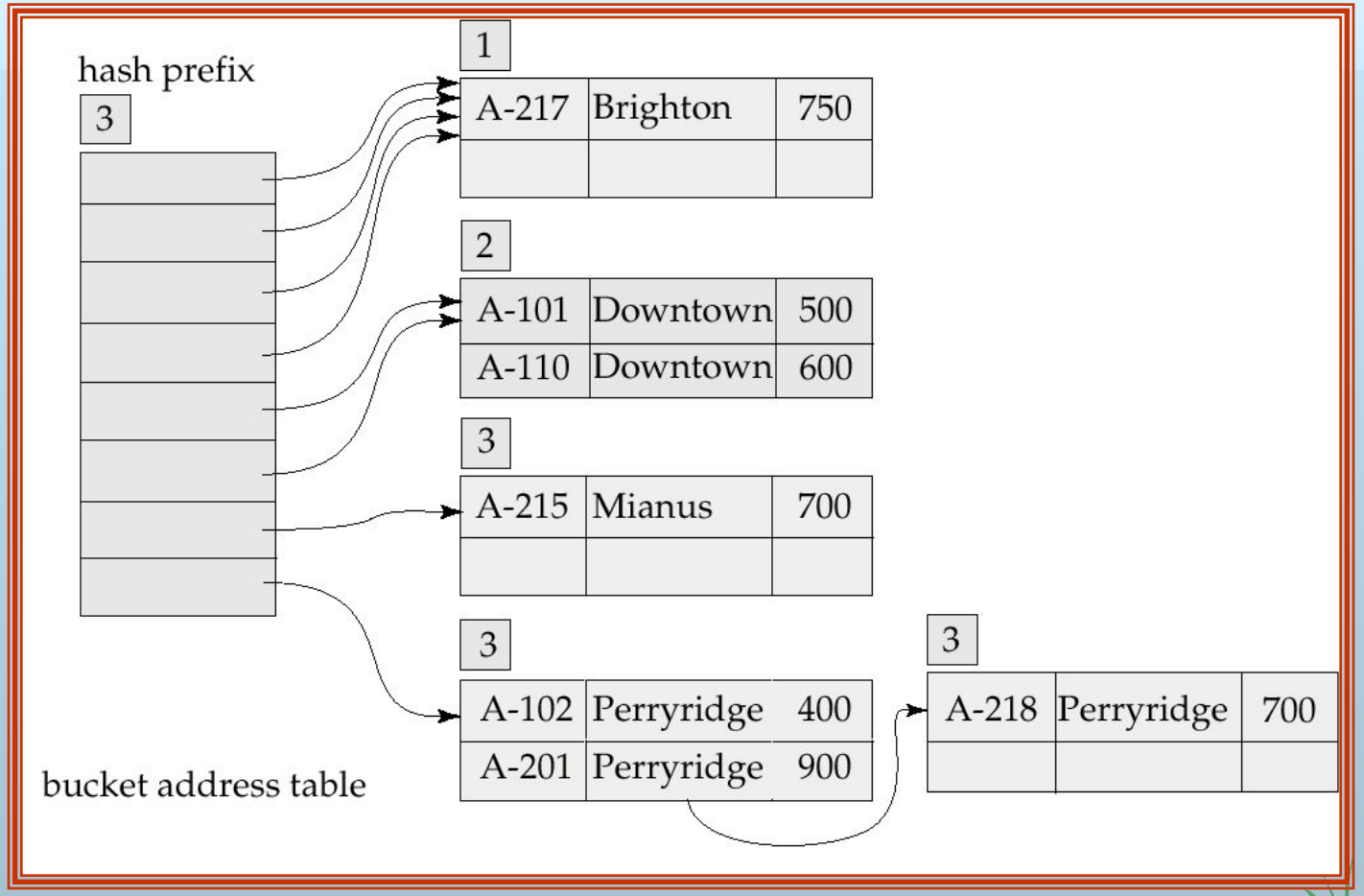


Example (Cont.)

Hash structure after insertion of Mianus record



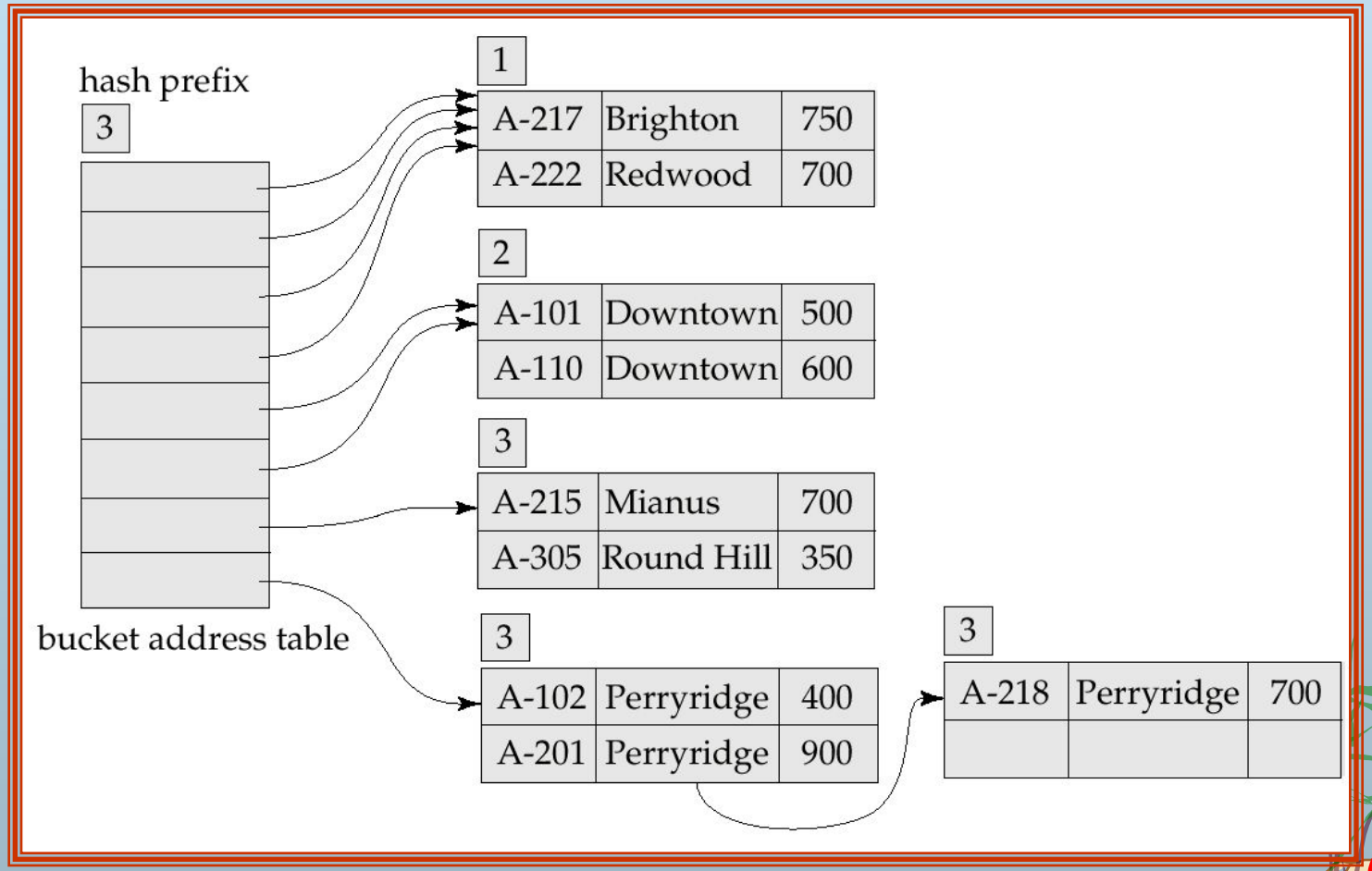
Example (Cont.)



Hash structure after insertion of three Perryridge records

Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - ★ Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch-name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  select *account-number*

  from *account*

  where *branch-name* = “Perryridge” **and** *balance* = 1000

- Possible strategies for processing query using indices on single attributes:
  1. Use index on *branch-name* to find accounts with balances of $1000; test *branch-name* = “Perryridge”.
  2. Use index on *balance* to find accounts with balances of $1000; test *branch-name* = “Perryridge”.
  3. Use *branch-name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

Indices on Multiple Attributes

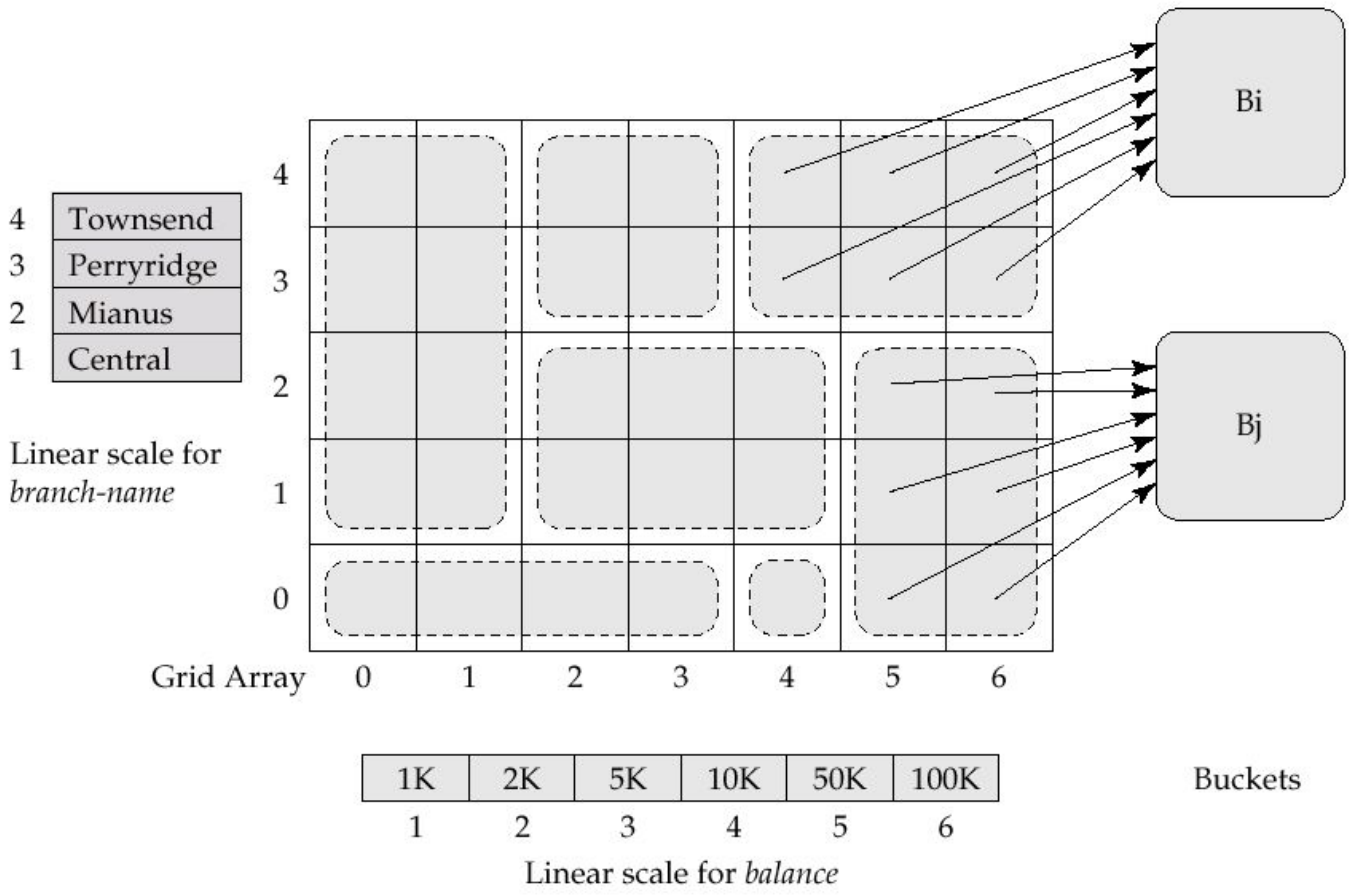
Suppose we have an index on combined search-key
(*branch-name, balance*).

- With the **where** clause
  where *branch-name* = “Perryridge” **and** *balance* = 1000
  the index on the combined search-key will fetch only records that satisfy both conditions.
  Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *branch-name* = “Perryridge” **and** *balance* < 1000
- But cannot efficiently handle
  where *branch-name* < “Perryridge” **and** *balance* = 1000
  May fetch many records that satisfy the first but not the second condition.

Grid Files

- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.
- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

Example Grid File for *account*



Queries on a Grid File

- A grid file on two attributes A and B can handle queries of all following forms with reasonable efficiency
  - $(a_1 \le A \le a_2)$
  - $(b_1 \le B \le b_2)$
  - $(a_1 \le A \le a_2 \ \wedge \ b_1 \le B \le b_2)$,.
- E.g., to answer $(a_1 \le A \le a_2 \ \wedge \ b_1 \le B \le b_2)$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

Grid Files (Cont.)

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.
  - Idea similar to extendable hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many overflow buckets.
- Periodic re-organization to increase grid size will help.
  - But reorganization can be very expensive.
- Space overhead of grid array can be high.
- R-trees (Chapter 23) are an alternative

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - ★ Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*name*	*gender*	*address*	*income-level*
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income-level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

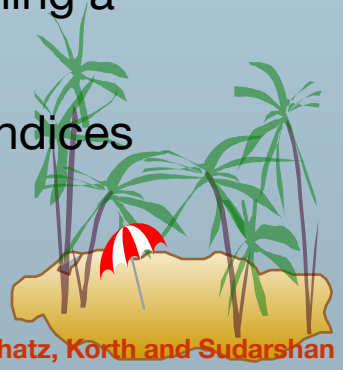
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - ★ Can then retrieve required tuples.
    - ★ Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A=v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A=v*):
    - intersect above result with (NOT *bitmap-A-Null*)

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - E.g. 1-million-bit maps can be anded with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - ★ Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B^+-tree indices



End of Chapter

Partitioned Hashing

- Hash values are split into segments that depend on each attribute of the search-key.

  $(A_1, A_2, \ldots, A_n)$ for n attribute search-key

- Example: $n = 2$, for *customer,* search-key being (*customer-street, customer-city*)

search-key value	*hash value*
(Main, Harrison)	101 111
(Main, Brooklyn)	101 001
(Park, Palo Alto)	010 010
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010

- To answer equality query on single attribute, need to look up multiple buckets. Similar in effect to grid files.

Module 17: Transactions

Database System Concepts, 7th Ed.

Outline

- Transaction Concept
- Transaction State
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

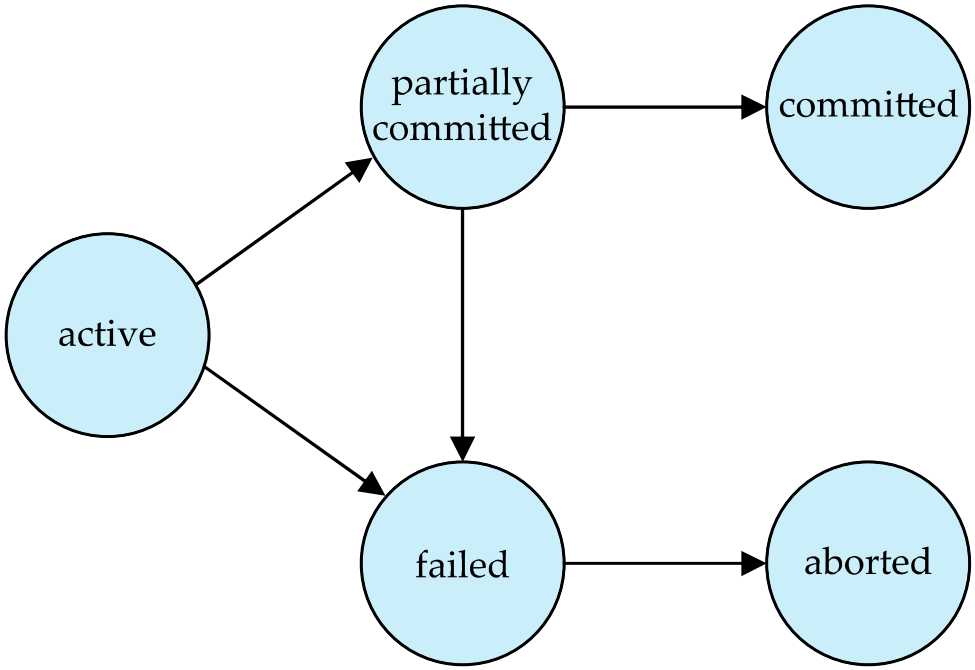
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - Can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **Conflict serializability**
  2. **View serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q,*
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Other Notions of Serializability

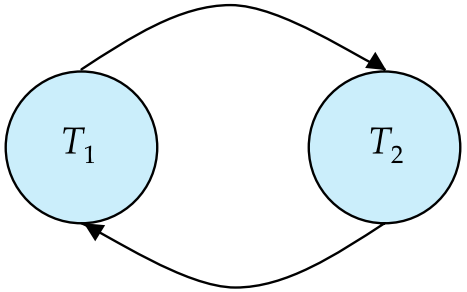
- The schedule below produces same outcome as the serial schedule $< T_1, T_5 >$, yet is not conflict equivalent or view equivalent to it.

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

- Determining such equivalence requires analysis of operations other than read and write.

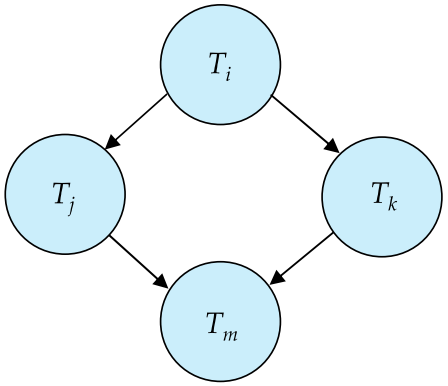
Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- Example of a precedence graph

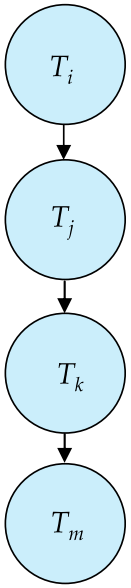


Test for Conflict Serializability

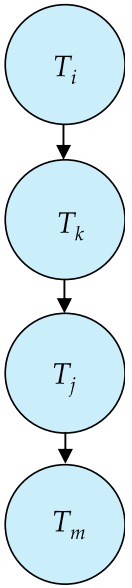
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

Need to address the effect of transaction failures on concurrently running transactions.

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur;
  - For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every Cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control (Cont.)

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
    - E.g. In SQL **set transaction isolation level serializable**
    - E.g. in JDBC -- connection.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Transactions as SQL Statements

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu's salary = 90000
  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'
- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

End of Chapter 17

View Serializability

T1	T2
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
	B:= B + temp
	write(B)

Schedule-1

Are these two schedules View Equivalent?

T1	T2
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
	B:= B + temp
	write(B)
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	

Schedule-2

T1	T2
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
	B:= B + temp
	write(B)

Schedule-1

These two schedules are not View Equivalent

T1	T2
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
	B:= B + temp
	write(B)
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	

Schedule-2

T1	T2
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
	B:= B + temp
	write(B)

Schedule-1

Are these two schedules View Equivalent?

T1	T2
read(A)	
A:= A – 50	
write(A)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B + temp
	write(B)

Schedule-3

T1	T2
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
	B:= B + temp
	write(B)

Schedule-1

These two schedules are View Equivalent

T1	T2
read(A)	
A:= A – 50	
write(A)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B + temp
	write(B)

Schedule-3

Note:

- A schedule *S* is view serializable, if it is view equivalent to a serial schedule.
- Here, **Schedule-3 is View Serializable**. [because, this schedule is view equivalent to a serial schedule (Schedule-1)]
- Every conflict serializable schedule is also view serializable.
- Here, **Schedule-3 is both View Serializable and Conflict Sesrializable.**

Every conflict serializable schedule is also view serializable, but there are view serializable schedules that are not conflict serializable.

T3	T4	T6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Schedule-9

Are these two schedules View Equivalent?

Are these two schedules Conflict Equivalent?

T3	T4	T6
read(Q)		
write(Q)		
	write(Q)	
		write(Q)

T3	T4	T6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Schedule-9

These two schedules are View Equivalent but not Conflict Equivalent

T3	T4	T6
read(Q)		
write(Q)		
	write(Q)	
		write(Q)

- Here, **Schedule-9 is View Serializable but not Conflict Sesrializable.**

Note:

Every view serializable schedule that is not conflict serializable has blind writes.

Other Examples

T1	T5
read(A)	
A:= A – 50	
write(A)	
	read(B)
	B:= B – 10
	write(B)
read(B)	
B:= B + 50	
write(B)	
	read(A)
	A:= A + 10
	write(A)

Schedule-8

Are these two schedules View Equivalent?

Are these two schedules Conflict Equivalent?

T1	T5
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B – 10
	write(B)
	read(A)
	A:= A + 10
	write(A)

T1	T5
read(A)	
A:= A – 50	
write(A)	
	read(B)
	B:= B – 10
	write(B)
read(B)	
B:= B + 50	
write(B)	
	read(A)
	A:= A + 10
	write(A)

Schedule-8

These two schedules produce same outcome, but neither Conflict Equivalent nor View Equivalent

T1	T5
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B – 10
	write(B)
	read(A)
	A:= A + 10
	write(A)

Conflict Serializability

T1	T2
read(A)	
A:= A – 50	
write(A)	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B + temp
	write(B)

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Fig: Schedule-3 (consistent schedule)

T1	T2
read(A)	
A:= A – 50	
	read(A)
	temp := A*0.1
	A:= A – temp
	write(A)
	read(B)
write(A)	
read(B)	
B:= B + 50	
write(B)	
	B:= B + temp
	write(B)

T1	T2
read(A)	
	read(A)
	write(A)
	read(B)
write(A)	
read(B)	
write(B)	
	write(B)

Fig: Schedule-4 (inconsistent schedule)

Conflict Equivalent and Serializable

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule-3

Swap non-conflicting instructions read(B) of T1 with write(A) of T2

T1	T2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Swap non-conflicting instructions read(B) of T1 with read(A) of T2

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

Swap non-conflicting instructions write(B) of T1 with write(A) of T2

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Swap non-conflicting instructions write(B) of T1 with read(A) of T2

T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule-6 (Serial schedule)

Here, Schedule-3 is Conflict Serializable

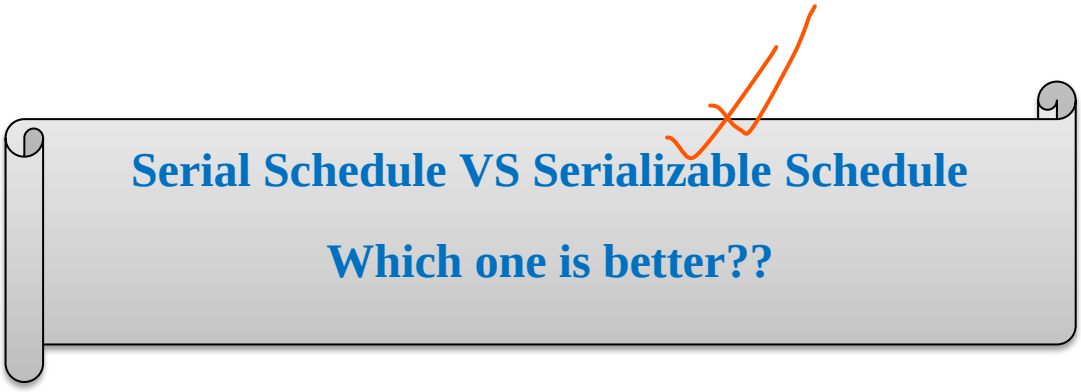
Schedule-3 and Schedule-6 is Conflict Equivalent

Summary:

- **Conflict Serializable:**
  - A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
  - We can say that a **schedule *S* is conflict serializable if it is conflict equivalent to a serial schedule**

- **Conflicting operations:** Two operations are said to be conflicting if all conditions satisfy:
  - They belong to different transactions
  - They operate on the same data item
  - At Least one of them is a write operation

- **Conflict Equivalent*:*** If a schedule *S* can be transformed into a schedule *S´* by a series of **swaps of non-conflicting instructions**, we say that *S* and *S´* are conflict equivalent.



Some more examples:

Example-1

T3	T4
read(Q)	
	write (Q)
write (Q)	

Fig: Schedule-7

Is Schedule-7 Conflict Serializable?????

T3	T4
read(Q)	
	write (Q)
write (Q)	

Fig: Schedule-7 (not conflict serializable)

Example-2

**It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.

T1	T5
read(A)	
A:= A – 50	
write(A)	
	read(B)
	B:= B – 10
	write(B)
read(B)	
B:= B + 50	
write(B)	
	read(A)
	A:= A + 10
	write(A)

Are these two Schedules Consistant??????

Are these two Schedules Conflict Equivalent??????

T1	T5
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B – 10
	write(B)
	read(A)
	A:= A + 10
	write(A)

T1	T5
read(A)	
A:= A – 50	
write(A)	
	read(B)
	B:= B – 10
	write(B)
read(B)	
B:= B + 50	
write(B)	
	read(A)
	A:= A + 10
	write(A)

These two schedules produce same outcome, but not Conflict Equivalent

Schedule-8 is not Conflict Serializable.

T1	T5
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
	read(B)
	B:= B – 10
	write(B)
	read(A)
	A:= A + 10
	write(A)

Fig: Schedule-8

Chapter 18 : Concurrency Control

Database System Concepts, 7th Ed.

Outline

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Insert and Delete Operations
- Concurrency in Index Structures

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Schedule With Lock Grants

- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?)
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Deadlock

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

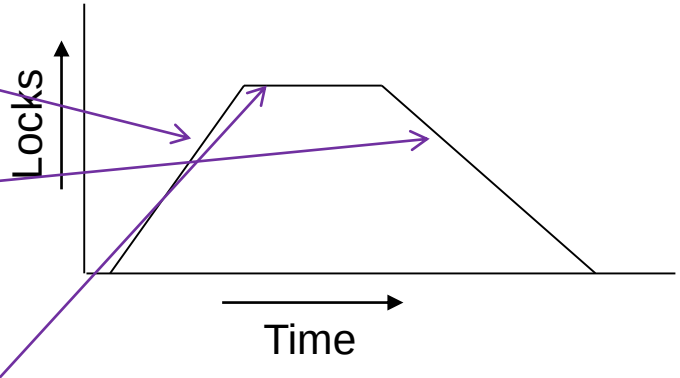
- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Deadlock (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).



The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability of freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary for conflict serializability *in the following sense*:
  - *Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.*

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display($A + B$)
lock-X(A)	
read(A)	
$A := A + 50$	
write(A)	
unlock(A)	

Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can **convert** a lock-S to a lock-X (**upgrade**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

```
if Ti has a lock on D
  then
        read(D)
  else begin
        if necessary wait until no other
           transaction has a lock-X on D
        grant Ti a  lock-S on D;
        read(D)
       end
```

Automatic Acquisition of Locks (Cont.)

- The operation **write**(*D*) is processed as:

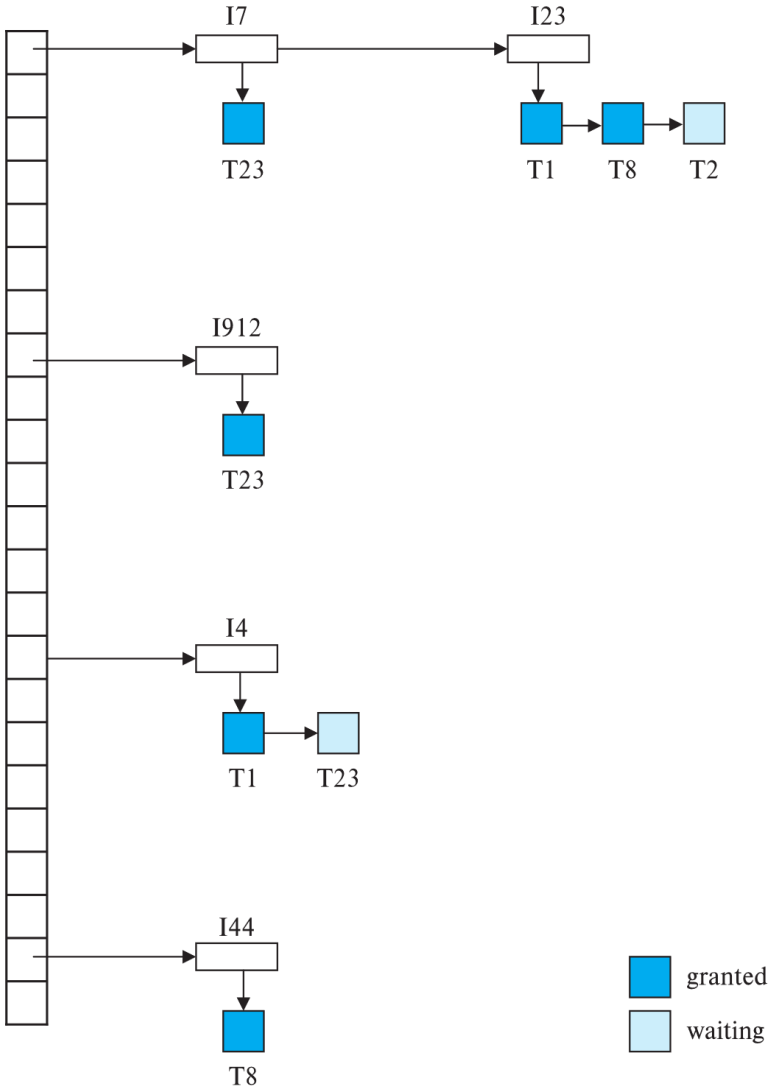
```
if Ti has a  lock-X on D
  then
    write(D)
 else begin
     if necessary wait until no other trans. has any lock on D,
     if Ti has a lock-S on D
          then
             upgrade lock on D  to lock-X
         else
             grant Ti a lock-X on D
         write(D)
  end;
```

- **All locks are released after commit or abort**

Implementation of Locking

- A **lock manager** can be implemented as a separate process
- Transactions can send lock and unlock requests as messages
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
  - The requesting transaction waits until its request is answered
- The lock manager maintains an in-memory data-structure called a **lock table** to record granted locks and pending requests

Lock Table



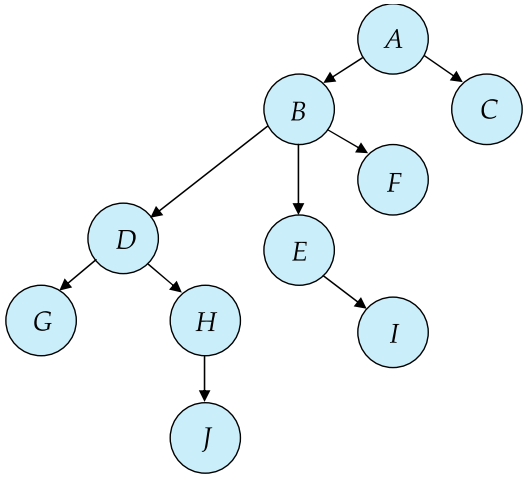
- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = {d_1, d_2 ,..., d_h} of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

- Only exclusive locks are allowed.
- The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
- Data items may be unlocked at any time.
- A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - Shorter waiting times, and increase in concurrency
  - Protocol is deadlock-free, no rollbacks are required
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under the tree protocol, and vice versa.

Deadlock Handling

- System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Deadlock Handling

Two principle scheme:

1. *Deadlock prevention*
2. *Deadlock detection and recovery*

*** Timeout based scheme is another way of handling deadlock that falls somewhere between deadlock prevention; and detection and recovery.*

Deadlock Handling

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. There are two approaches to deadlock prevention.

- **First Approach:** It ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.
- **Second Approach:** It is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.(Using preemption and transaction rollbacks).

Deadlock Handling

- ***Deadlock prevention*** (First approach)

First scheme under First approach: It requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked.

There are two main disadvantages to this protocol:

(1) it is often hard to predict, before the transaction begins, what data items need to be locked;

(2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Deadlock Handling

- ***Deadlock prevention*** (First approach)

Second scheme under First approach:

- To impose **partial ordering** of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).
- A variation of this approach is to use a **total order** of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that **precede** that item in the ordering. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed it to ensure that locks are requested in the right order.

Deadlock Prevention (2nd Approach)

- The second approach for preventing deadlocks is to use **preemption and transaction rollbacks.**
- In preemption, when a transaction T2 requests a lock that transaction T1 holds, the lock granted to T1 may be preempted by rolling back of T1, and granting of the lock to T2.
- To control the preemption, a unique timestamp is assigned to each transaction. The system uses these timestamps only to decide whether a transaction should wait or roll back.

Deadlock Prevention (2nd Approach)

- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with **its original** timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

Deadlock Prevention (2nd Approach)

- **Rollbacks in wait-die and wound-wait**

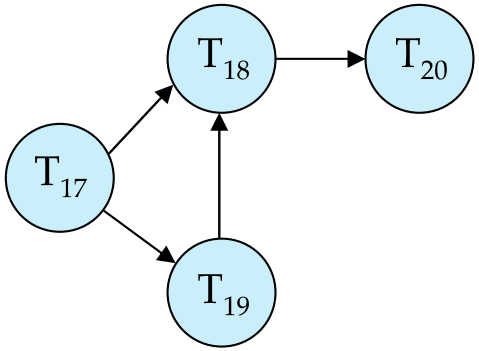
In the wait–die scheme, if a transaction Ti dies and is rolled back because it requested a data item held by transaction Tj, then Ti may reissue the same sequence of requests when it is restarted. If the data item is still held by Tj , then Ti will die again. Thus, Ti may die several times before acquiring the needed data item. Contrast this series of events with what happens in the wound–wait scheme. Transaction Ti is wounded and rolled back because Tj requested a data item that it holds. When Ti is restarted and requests the data item now being held by Tj , Ti waits. **Thus, there may be fewer rollbacks in the wound–wait scheme.**

Deadlock Handling(Cont.)

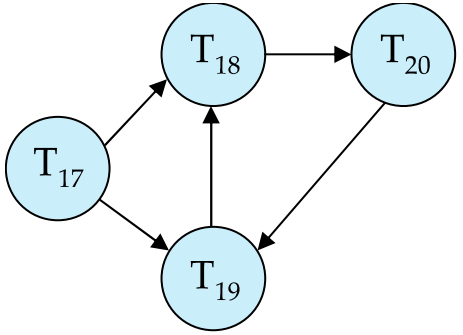
- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

- Selection of a victim
- Rollback
- Starvation

Deadlock Recovery

Selection of a victim

- Choose the ones that will incur minimum cost.
- Many factors may determine the cost of a rollback, including
  - How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
  - How many data items the transaction has used.
  - How many more data items the transaction needs for it to complete.
  - How many transactions will be involved in the rollback.

Deadlock Recovery

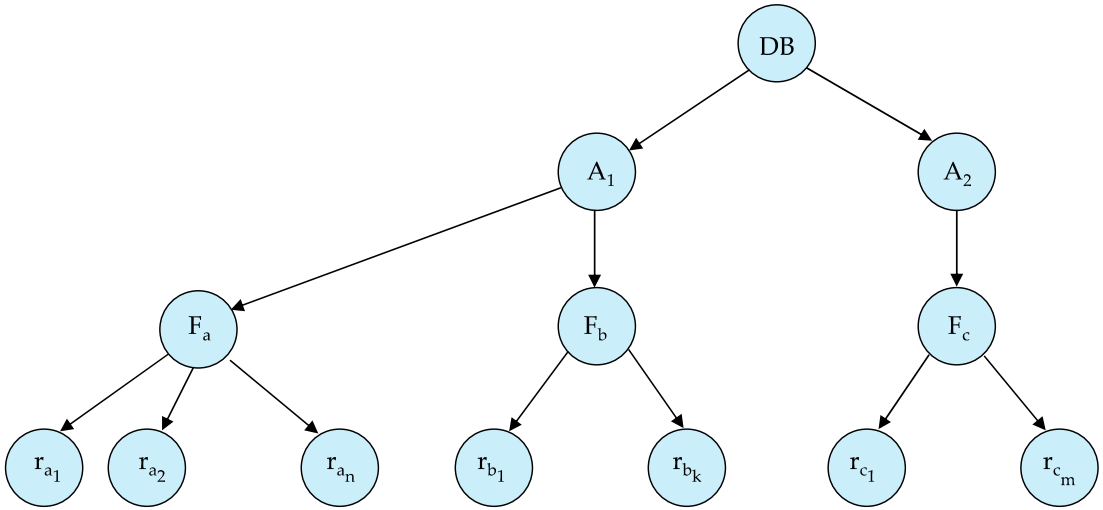
- Rollback
  - determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim
  - Another is to include the number of rollbacks in the cost factor.

Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree ***explicitly*****,** it *implicitly* locks all the **node's descendants** in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): high concurrency, high locking overhead
  - **Coarse granularity** (higher in tree): low locking overhead, low concurrency

Example of Granularity Hierarchy

- The levels, starting from the coarsest (top) level are
  - *database*
  - *area*
  - *file*
  - *record*
- The corresponding tree



Multiple Granularity(Intention Lock Modes)

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Why Intention Lock Modes?

- Suppose that transaction Tj wishes to lock record rb6 of file Fb. Since Ti has locked Fb explicitly, it follows that rb6 is also locked (implicitly). But, when Tj issues a lock request for rb6 , rb6 is not explicitly locked!

- How does the system determine whether Tj can lock rb6?
  - Tj must traverse the tree from the root to record rb6 . If any node in that path is locked in an incompatible mode, then Tj must be delayed.

Why Intention Lock Modes?

- Suppose now that transaction Tk wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that Tk should not succeed in locking the root node, since Ti is currently holding a lock on part of the tree (specifically, on file Fb).
- But how does the system determine if the root node can be locked?
  - One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme.
- A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**.
- Intention locks are put on **all the ancestors** of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully.

Why Intention Lock Modes?

- Suppose that transaction Tj wishes to lock record rb6 of file Fb. Since Ti has locked Fb explicitly, it follows that rb6 is also locked (implicitly). But, when Tj issues a lock request for rb6 , rb6 is not explicitly locked!

- How does the system determine whether Tj can lock rb6?
  - Tj must traverse the tree from the root to record rb6 . If any node in that path is locked in an incompatible mode, then Tj must be delayed.

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Protocol

- Transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
  4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Multiple Granularity Locking Protocol

As an illustration of the protocol, consider the granularity hierarchy and these transactions:

- Suppose that transaction T18 reads record ra2 in file Fa. Then, T18 needs to lock the database, area A1, and Fa in IS mode (and in that order), and finally to lock ra2 in S mode.
- Suppose that transaction T19 modifies record ra9 in file Fa. Then, T19 needs to lock the database, area A1, and file Fa in IX mode, and finally to lock ra9 in X mode.
- Suppose that transaction T20 reads all the records in file Fa. Then, T20 needs to lock the database and area A1 (in that order) in IS mode, and finally to lock Fa in S mode.
- Suppose that transaction T21 reads the entire database. It can do so after locking the database in S mode.

Multiple Granularity

- Ensures Serializability
- This protocol enhances concurrency and reduces lock overhead.
- It is particularly useful in applications that include a mix of
  - Short transactions that access only a few data items
  - Long transactions that produce reports from an entire file or set of files
- Deadlock is possible in the protocol as it is in the two-phase locking protocol.

Insert/Delete Operations and Predicate Reads

- Locking rules for insert/delete operations
  - An exclusive lock must be obtained on an item before it is deleted
  - A transaction that inserts a new tuple into the database I automatically given an X-mode lock on the tuple
- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits

Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - **select count**(*)
      from *instructor*
      where *dept_name* = 'Physics'
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - **insert into** *instructor* **values** ('11111', 'Feynman', 'Physics', 94000)

    (conceptually) conflict in spite of not accessing any tuple in common.
- If only tuple locks are used, non-serializable schedules can result
  - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction
- Can also occur with updates
  - E.g. update Wu's department from Finance to Physics

Insert/Delete Operations and Predicate Reads

- **Another Example**: T1 and T2 both find maximum instructor ID in parallel, and create new instructors with ID = maximum ID + 1
  - Both instructors get same ID, not possible in serializable schedule
- Schedule

T1	T2
Read(instructor where dept_name='Physics')	
	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
Read(instructor where dept_name='Comp. Sci.')	

Handling Phantoms

- There is a conflict at the data level
  - The transaction performing predicate read or scanning the relation is reading information that indicates what tuples the relation contains
  - The transaction inserting/deleting/updating a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very low concurrency for insertions/deletions.

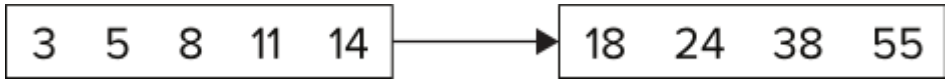
Index Locking To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation
  - A transaction T_i that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction T_i that inserts, updates or deletes a tuple t_i in a relation r
    - Must update all indices to r
    - Must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

Next-Key Locking to Prevent Phantoms

- Index-locking protocol to prevent phantoms locks entire leaf node
  - Can result in poor concurrency if there are many inserts
- **Next-key locking protocol**: provides higher concurrency
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
    - even for inserts/deletes
  - Lock mode: S for lookups, X for insert/delete/update
- Ensures detection of query conflicts with inserts, deletes and updates

Consider B+-tree leaf nodes as below, with query predicate $7 \leq X \leq 16$.
Check what happens with next-key locking when inserting: (i) 15 and (ii) 7



Timestamp Based Concurrency Control

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be implemented using 2 methods:
    - Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
    - Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.
- Timestamp-based protocols manage concurrent execution such that
  time-stamp order = serializability order
- Several alternative protocols based on timestamps (TSO , Thomas Write Rule.)

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- Maintains for each data *Q* two timestamp values:
  - **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_i issues a **read**(Q)

  1. If TS(T_i) < **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back.
  2. If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to

     max(R-timestamp(Q), TS(T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously.

  Hence, the **write** operation is rejected, and T_i is rolled back.

  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Example of Schedule Under TSO

- Is this schedule valid under TSO?

  Assume that initially:
  R-TS(A) = W-TS(A) = 0
  R-TS(B) = W-TS(B) = 0
  Assume TS(T_{25}) = 25 and
  TS(T_{26}) = 26

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

- How about this one, where initially
  R-TS(Q)=W-TS(Q)=0

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

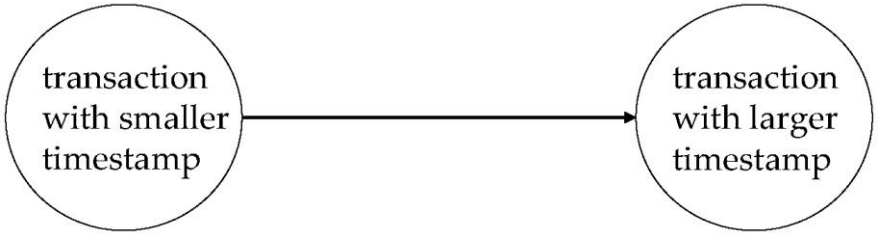
Another Example Under TSO

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all R-TS and W-TS = 0 initially

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.
- But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2:
  - Limited form of locking: wait for data to be committed before reading it
- Solution 3:
  - Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done, this {**write**} operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

Idea:

- Postpone writes to end of transaction
  - Keep track of data items read/written by transaction
  - **Validation** performed at commit time, detect any out-of-serialization order reads/writes
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.
  1. **Read and execution phase**: Transaction T_i writes only to temporary local variables
  2. **Validation phase**: Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.
  3. **Write phase**: If T_i is validated, the updates are applied to the database; otherwise, T_i is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
  - We assume for simplicity that the validation and write phase occur together, atomically and serially
    - I.e., only one transaction executes validation/write at a time.

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - **StartTS**(T_i) : the time when T_i started its execution
  - **ValidationTS**(T_i): the time when T_i entered its validation phase
  - **FinishTS**(T_i) : the time when T_i finished its write phase
- Validation tests use above timestamps and read/write sets to ensure that serializability order is determined by validation time
  - Thus, $TS(T_i) = ValidationTS(T_i)$
- Validation-based protocol has been found to give greater degree of concurrency than locking/TSO if probability of conflicts is low.

Validation Test for Transaction T_j

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finishTS**(T_i) < **startTS**(T_j)
  - **startTS**(T_j) < **finishTS**(T_i) < **validationTS**(T_j) **and** the set of data items **written by T_i does not intersect with the set of data items read by T_j.**

  then validation succeeds and T_j can be committed.
- Otherwise, validation fails and T_j is aborted.
- Justification:
  - First condition applies when execution is not concurrent
    - The writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - If the second condition holds, execution is concurrent, T_j does not read any item written by T_i.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
<validate>	
display($A + B$)	
	<validate>
	write(B)
	write(A)

Multiversion Concurrency Control

Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**
- Key ideas:
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction issuing the read request, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $<Q_1, Q_2, ..., Q_m>$. Each version Q_k contains three data fields:
  - **Content** -- the value of version Q_k.
  - **W-timestamp**(Q_k) -- timestamp of the transaction that created (wrote) version Q_k
  - **R-timestamp**(Q_k) -- largest timestamp of a transaction that successfully read version Q_k

Multiversion Timestamp Ordering (Cont)

- Suppose that transaction T_i issues a **read**(Q) or **write**(Q) operation. Let $\boldsymbol{Q_k}$ denote the version of Q whose write timestamp is the **largest write timestamp less than or equal to TS(T_i).**
  1. If transaction T_i issues a **read**(Q), then
     - the value returned is the content of version Q_k
     - If R-timestamp(Q_k) < TS(T_i), set R-timestamp(Q_k) = TS(T_i),
  2. If transaction T_i issues a **write**(Q)
     1. if TS(T_i) < R-timestamp(Q_k), then transaction T_i is rolled back.
     2. if TS(T_i) = W-timestamp(Q_k), the contents of Q_k are overwritten
     3. Otherwise, a new version Q_i of Q is created
        - W-timestamp(Q_i) and R-timestamp(Q_i) are initialized to TS(T_i).

Multiversion Timestamp Ordering (Cont)

- Observations
  - Reads always succeed
  - A write by T_i is rejected if some other transaction T_j that (in the serialization order defined by the timestamp values) should read T_i's write, has already read a version created by a transaction older than T_i.
- Protocol guarantees serializability

Multiversion Two-Phase Locking

- Differentiates between read-only transactions and update transactions
- **Update transactions** acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Read of a data item returns the latest version of the item
  - The first **write** of Q by T_i results in the creation of a new version Q_i of the data item Q written
    - W-timestamp(Q_i) set to ∞ initially
  - When update transaction T_i completes, commit processing occurs:
    - Value **ts-counter** stored in the database is used to assign timestamps
      - **ts-counter** is locked in two-phase manner
    - Set TS(T_i) = **ts-counter** + 1
    - Set W-timestamp(Q_i) = TS(T_i) for all versions Q_i that it creates
    - **ts-counter** = **ts-counter +** 1

Multiversion Two-Phase Locking (Cont.)

- **Read-only transactions**
  - are assigned a timestamp = **ts-counter** when they start execution
  - follow the **multiversion timestamp-ordering protocol** for performing reads
    - Do not obtain any locks
- Read-only transactions that start after T_i increments **ts-counter** will see the values updated by T_i.
- Read-only transactions that start before T_i increments the **ts-counter** will see the value before the updates by T_i.
- Only serializable schedules are produced.

MVCC: Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - E.g., if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > 9, than Q5 will never be required again
- Issues with
  - primary key and foreign key constraint checking
  - Indexing of records with multiple versions

Snapshot Isolation

- Motivation: Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results
- Solution 1: Use multiversion 2-phase locking
  - Give logical "snapshot" of database state to read only transaction
    - Reads performed on snapshot
  - Update (read-write) transactions use normal locking
  - Works well, but how does system know a transaction is read only?
- Solution 2 (partial): Give snapshot of database state to every transaction
  - Reads performed on snapshot
  - Use 2-phase locking on updated data items
  - Problem: variety of anomalies such as lost update can result
  - Better solution: snapshot isolation level (next slide)

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible to T1
  - Writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

T_1 deposits 50 in Y	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

T_1 deposits 50 in X	T_2 withdraws 50 from X
$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

- Variant: "**First-updater-wins**"
  - Check for concurrent updates when write occurs by locking item
    - But lock should be held till all concurrent transactions have finished
  - (Oracle uses this plus some extra features)
  - Differs only in when abort occurs, otherwise equivalent

Benefits of SI

- Reads are *never* blocked,
  - and also don't block other txns activities
- Performance similar to Read Committed
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - No lost update
    - I.e., update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - I.e., if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent txns, one sees the effects of the other
    - In SI: neither sees the effects of the other
  - Result: Integrity constraints can be violated

Snapshot Isolation

- Example of problem with SI
  - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ??
    - if both transactions start at the same time, with snapshot isolation: A = ?? , B = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1
    - Two transaction can both create order with same number
      - Is an example of phantom phenomenon

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Snapshot Isolation Anomalies

- SI breaks serializability when transactions modify *different* items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
    - when txns conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
  - But problems do occur
    - Application developers should be careful about write skew
- SI can also cause a read-only transaction anomaly, where read-only transaction may see an inconsistent state even if updaters are serializable
  - We omit details
- Using snapshots to verify primary/foreign key integrity can lead to inconsistency
  - Integrity constraint checking usually done outside of snapshot

Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
- Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts
  - Where T_i writes a data a data item Q, T_j reads an earlier version of Q, but T_j is serialized after T_i
- Idea: track read-write dependencies separately, and roll-back transactions where cycles can occur
  - Ensures serializability
  - Details in book
- Implemented in PostgreSQL from version 9.1 onwards
  - PostgreSQL implementation of SSI also uses index locking to detect phantom conflicts, thus ensuring true serializability

SI Implementations

- Snapshot isolation supported by many databases
  - Including Oracle, PostgreSQL, SQL Server, IBM DB2, etc
  - Isolation level can be set to snapshot isolation
- Oracle implements "first updater wins" rule (variant of "first committer wins")
  - Concurrent writer check is done at time of write, not at commit time
  - Allows transactions to be rolled back earlier
- **Warning**: *even if isolation level is set to serializable, Oracle actually uses snapshot isolation*
  - Old versions of PostgreSQL prior to 9.1 did this too
  - Oracle and PostgreSQL < 9.1 do not support true serializable execution

Working Around SI Anomalies

- Can work around SI anomalies for specific queries by using **select .. for update** (supported e.g. in Oracle)
  - Example
    - **select max**(orderno) **from** orders **for update**
    - read value into local variable maxorder
    - insert into orders (maxorder+1, …)
- **select for update (SFU) clause** treats all data read by the query as if it were also updated, preventing concurrent updates
- Can be added to queries to ensure serializability in many applications
  - Does not handle phantom phenomenon/predicate reads though

Weak Levels of Concurrency

Weak Levels of Consistency

- **Degree-two consistency**: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- **Cursor stability**:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction
  - Special case of degree-two consistency

Weak Levels of Consistency in SQL

- SQL allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even uncommitted data to be read
- In most database systems, read committed is the default consistency level
  - Can be changed as database configuration parameter, or per transaction
    - **set isolation level serializable**

Concurrency Control across User Interactions

- Many applications need transaction support across user interactions
  - Can’t use locking for long durations
- Application level concurrency control
  - Each tuple has a version number
  - Transaction notes version number when reading tuple
    - **select** r.balance, r.version **into** :A, :version
      from r **where** acctId =23
  - When writing tuple, check that current version number is same as the version when tuple was read
    - **update** r **set** r.balance = r.balance + :deposit, r.version = r.version+1
      where acctId = 23 **and** r.version = :version

Concurrency Control across User Interactions

- Equivalent to **optimistic concurrency control without validating read set**
  - Unlike SI, reads are not guaranteed to be from a single snapshot.
  - Does not guarantee serializability
  - But avoids some anomalies such as “lost update anomaly”
- Used internally in Hibernate ORM system
- Implemented manually in many applications
- Version numbers stored in tuples can also be used to support first committer wins check of snapshot isolation

Advanced topics in Concurrency Control

Online Index Creation

- Problem: how to create an index on a large relation without affecting concurrent updates
  - Index construction may take a long time
  - Two-phase locking will block all concurrent updates
- Key ideas:
  - Build index on a snapshot of the relation, but keep track of all updates that occur after snapshot
    - Updates are not applied on the index at this point
  - Then apply subsequent updates to catch up
  - Acquire relation lock towards end of catchup phase to block concurrent updates
  - Catch up with remaining updates, and add index to system catalog
  - Subsequent transactions will find the index in catalog and update it

Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, e.g. by 2-phase locking of index nodes can lead to low concurrency.
- There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node.

Concurrency in Index Structures (Cont.)

- **Crabbing protocol** used instead of two-phase locking on the nodes of the B^+-tree during search/insertion/deletion:
  - First lock the root node in shared mode.
  - After locking all required children of a node in shared mode, release the lock on the node
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Above protocol can cause excessive deadlocks
  - Searches coming down the tree deadlock with updates going up the tree
  - Can abort and restart search, without affecting transaction
- The **B-link tree locking protocol** improves concurrency
  - Intuition: release lock on parent before acquiring lock on child
    - And deal with changes that may have happened between lock release and acquire

Concurrency Control in Main-Memory Databases

- Index locking protocols can be simplified with main-memory databases
  - Short term lock can be obtained on entire index for duration of an operation, serializing updates on the index
    - Avoids overheads of multiple lock acquire/release
    - No major penalty since operations finish fast, since there is no disk wait
- Latch-free techniques for data-structure update can speed up operations further

Latch-Free Data-structure Updates

- This code is not safe without latches if executed concurrently:

```
insert(value, head) {
    node = new node
    node->value = value
    node->next = head
    head = node
}
```

- This code is safe

```
insert latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        oldhead = head
        node->next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}
```

Latch-Free Data-structure Updates

- This code is not safe without latches if executed concurrently:

```
insert(value, head) {
    node = new node
    node->value = value
    node->next = head
    head = node
}
```

- This code is safe

```
insert latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        oldhead = head
        node->next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}
```

Latch-Free Data-structures (Cont.)

- *Consider:*

```
delete latchfree(head) {
    /* This function is not quite safe; see explanation in text. */
    repeat
        oldhead = head
        newhead = oldhead->next
        result = CAS(head, oldhead, newhead)
    until (result == success)
}
```

- Above code is almost correct, but has a concurrency bug
  - P1 initiates delete with N1 as head; concurrently P2 deletes N1 and next node N2, and then reinserts N1 as head, with N3 as next
  - P1 may set head as N2 instead of N3.
- Known as ABA problem
- See book for details of how to avoid this problem

Concurrency Control with Operations

- Consider this non-two phase schedule, which preserves database integrity constraints
- Can be understood as transaction performing increment operation
  - E.g., increment(A, -50), increment (B, 50)
  - As long as increment operation does not return actual value, increments can be reordered
    - ***Increments commute***
  - New increment-mode lock to support reordering
  - Conflict matrix with increment lock mode
    - *Two increment operations do not conflict with each other*

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Concurrency Control with Operations (Cont.)

- Undo of increment(v, n) is performed by increment (v, -n)
- Increment_conditional(v, n):
  - Updates v by adding n to it, as long as final v > 0, fails otherwise
  - Can be used to model, e.g. number of available tickets, *avail_tickets,* for a concert
  - Increment_conditional is NOT commutative
    - E.g., last few tickets for a concert
  - But reordering may still be acceptable

Real-Time Transaction Systems

- Transactions in a system may have deadlines within which they must be completed.
  - Hard deadline: missing deadline is an error
  - Firm deadline: value of transaction is 0 in case deadline is missed
  - Soft deadline: transaction still has some value if done after deadline
- Locking can cause blocking
- Optimistic concurrency control (validation protocol) has been shown to do will in a real-time setting

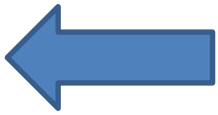
End of Chapter 18

Recoverability

T8	T9
read(A)	
write(A)	
	read(A)
write(A)	

Fig: Schedule-11

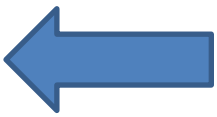
T8	T9
read(A)	
write(A)	
	read(A)
	commit
write(A)	
commit	



Is this schedule Recoverable/Irrecoverable?

Fig: Schedule-11

T8	T9
read(A)	
write(A)	
	read(A)
	commit
write(A)	
commit	

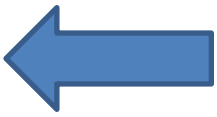


This schedule is Irrecoverable.

Fig: Schedule-11

T8	T9
read(A)	
write(A)	
	read(A)
write(A)	
commit	
	commit

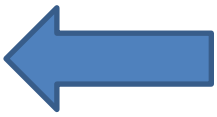
Fig: Schedule-11



Is this schedule now Recoverable?

T8	T9
read(A)	
write(A)	
	read(A)
write(A)	
commit	
	commit

Fig: Schedule-11



This schedule is now Recoverable.

T10	T11	T12
read(A)		
write(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Fig: Schedule-12

T10	T11	T12
read(A)		
write(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
commit		
	commit	
		commit

Fig: Schedule-12 (recoverable)

Note:

- Now Schedule-12 is recoverable but it cause Cascading Rollback.
- Cascading Rollback is undesirable.
- **Cascadeless Schedule:** for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj* .
- **Every cascadeless schedule is also recoverable.**

T10	T11	T12
read(A)		
write(B)		
write(A)		
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

Fig: Schedule-12 (cascadeless and recoverable)

Is a cascadeless schedule is a serial schedule?

T10	T11	T12
read(A)		
write(B)		
write(A)		
	write(A)	
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

Fig: Schedule-12 (cascadeless and recoverable)

Strict Schedule

T10	T11	T12
read(A)		
write(B)		
write(A)		
	write(A)	
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

Fig: Schedule-12 (cascadeless and recoverable)

T10	T11	T12
read(A)		
write(B)		
write(A)		
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

Fig: Schedule-12 (cascadeless, recoverable and strict)

Is a strict schedule is a serial schedule?

T10	T11	T12
read(A)		
write(B)		
write(A)		
	read(C)	
	write(C)	
commit		
	read(A)	
	write(A)	
	commit	
		read(A)
		commit

Fig: Schedule-12 (cascadeless, recoverable and strict)

Schedules:

1. **Recoverable**
2. **Cascadeless**
3. **Strict**

Note:

From Chapter-15 we have covered Slide#15.1-15.24

Transactions

CSE 301

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule-3

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule-1

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule-3

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule-2

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Schedule 4

T1	T2	T3
Read(A)		
	Read(B)	
	Write(A)	
Write(A)		
		Write(A)

Precedence Graph

Precedence graph — a directed graph which consists of a pair G=(V,E),

where the vertices are the transactions.

A schedule is conflict serializable if it’s precedence graph has no cycle in it.

Precedence Graph

The set of edges consists of all edges Ti →Tj for which one of three conditions holds:

1. Ti executes write(Q) before Tj executes read(Q).
2. Ti executes read(Q) before Tj executes write(Q).
3. Ti executes write(Q) before Tj executes write(Q).

Precedence Graph

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule-1

Precedence Graph

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule-3

Precedence Graph

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Schedule 4

Precedence Graph

T1	T2	T3
Read(A)		
	Read(B)	
	Write(A)	
Write(A)		
		Write(A)

Schedule 5

Precedence Graph

T1	T2	T3
R(A)		
R(B)		
	R(A)	
	R(C)	
W(B)		
		R(B)
		R(C)
		W(B)
	W(A)	
	W(C)	

Schedule 6

Precedence Graph

T1	T2	T3
R(A)		
		R(B)
		R(A)
	R(B)	
	R(C)	
		W(B)
	W(C)	
R(C)		
W(A)		
W(C)		

Schedule 7

Precedence Graph

T1	T2	T3	T4	T5
	read(X)			
read(Y) read(Z)				
				read(V) read(W) read(W)
	read(Y) write(Y)			
		write(Z)		
read(U)				
			read(Y) write(Y) read(Z) write(Z)	
read(U) write(U)				

Precedence Graph