



Chapter 10 : Concurrency Control



Textbook: Chapter 15



Chapter 10: Concurrency Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiversion Schemes
- Insert and Delete Operations



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.
 - Transactions **do not access** data items **before having acquired a lock** on that data item
 - Transactions **release** their locks on a data item only **after they have accessed a data item**



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 lock** on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.



Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

```
 $T_2$ : lock-S(A);  
read (A);  
unlock(A);  
lock-S(B);  
read (B);  
unlock(B);  
display(A+B)
```

- Locking as above is not sufficient to guarantee serializability — if A and B get updated in-between the read of A and B , the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.



Pitfalls of Lock-Based Protocols

- 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.



Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if the 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.
 - ▶ For instance, T1 is waiting for an X-lock, T2 has S-lock
 - ▶ T3 requests S-lock...then T4... (**can be granted**)
 - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control managers can be designed to prevent starvation.



The Two-Phase Locking Protocol

- This is 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).
- For instance, T3, T4 (two-phase), in T3, unlock-x(B) can be added just after lock-x(A)



The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking (S2PL)**. Here a transaction must hold **all its exclusive locks** till it commits/aborts.
- **Rigorous two-phase locking (SS2PL)** is even stricter: here *all locks* are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.



Lock Conversions

- Two-phase locking **with lock conversions**:
 - First 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)
 - Second Phase:
 - can release a lock-S
 - can release a lock-X
 - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.



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



Automatic Acquisition of Locks (Cont.)

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

if T_i has a **lock-X** on D

then

write(D)

else begin

 if necessary wait until no other trans. has any lock on D ,

 if T_i has a **lock-S** on D

then

 upgrade lock on D to **lock-X**

else

 grant T_i a **lock-X** on D

write(D)

end;

- All locks are released after commit or abort

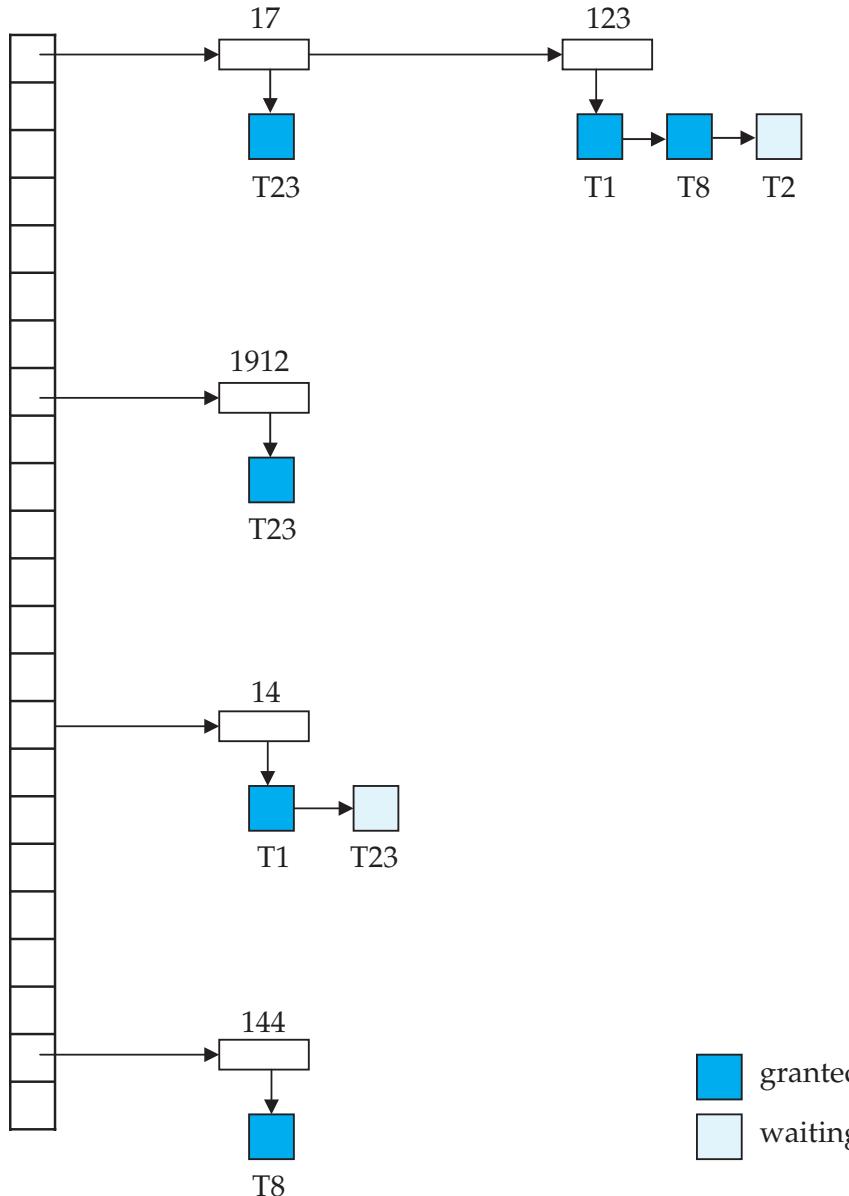


Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- 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 a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an **in-memory hash table** indexed on the name of the data item being locked



Lock Table



- Dark blue rectangles indicate granted locks, light blue 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



Deadlock Handling

- Consider the following two transactions:

T_1 : write (X)

write(Y)

T_2 : write(Y)

write(X)

- Schedule with deadlock

T_1	T_2
lock-X on A write (A) wait for lock-X on B	lock-X on B write (B) wait for lock-X on A



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.
- **Deadlock prevention** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
 - Require that **each transaction locks all its data items** before it begins execution (predeclaration).
 - ▶ Not practical
 - 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).



More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
 - **Preemptive:** Transaction holding a lock is aborted to make lock available
- **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. (T1 and T2, if T1 is younger, then T1 is rolled back-dies)
 - a transaction may die several times before acquiring needed data item (e.g., T2 does not need to wait for T1, if T1 is rolled back)
- **wound-wait** scheme — **preemptive**
 - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. (T1 and T2, if T1 is older, then T2 is rolled back)
 - may be fewer rollbacks than *wait-die* scheme.



Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its **original timestamp**. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- **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**.
 - thus deadlocks are not possible
 - simple to implement; but **starvation is possible**. Also difficult to determine good value of the timeout interval.

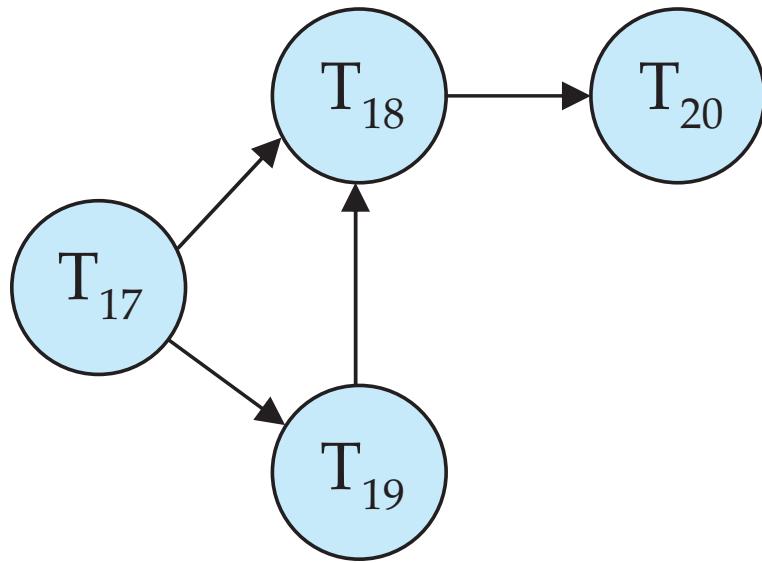


Deadlock Detection

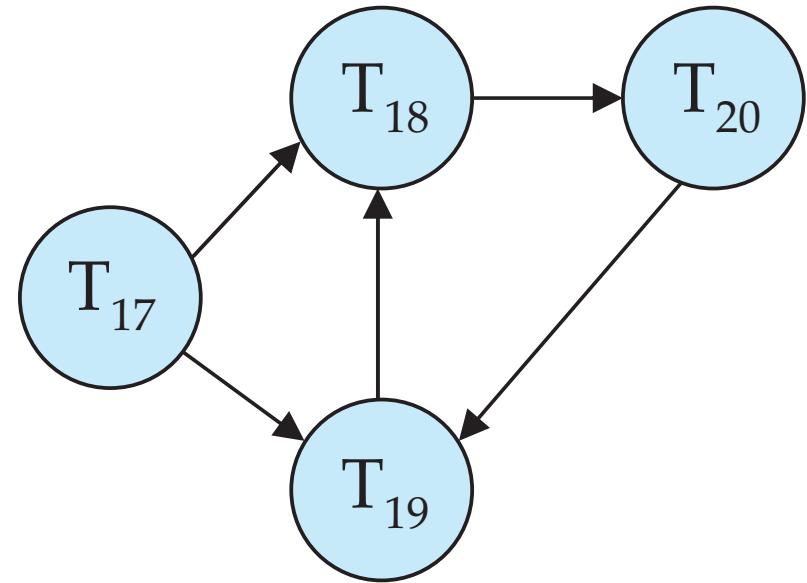
- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V, E)$,
 - V is a set of vertices (all the transactions in the system)
 - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E , then there is a directed edge from T_i to T_j , implying that **T_i is waiting for T_j to release a data item.**
- When T_i requests a data item currently being held by T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i .
- The system is in a deadlock state if and only if **the wait-for graph has a cycle**. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle



Deadlock Recovery

■ When deadlock is detected :

- Some transactions will have to be rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
- Rollback -- determine how far to roll back transaction
 - ▶ **Total rollback:** Abort the transaction and then restart it.
 - ▶ More effective to roll back transaction only **as far as necessary to break deadlock.**
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation



Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp $\text{TS}(T_i)$, a new transaction T_j is assigned time-stamp $\text{TS}(T_j)$ such that $\text{TS}(T_i) < \text{TS}(T_j)$.
- The protocol manages concurrent execution such that the **time-stamps determine the serializability order**.
- In order to assure such behavior, the 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.



Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)
 1. If $TS(T_i) \leq W\text{-timestamp}(Q)$, 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 $TS(T_i) \geq W\text{-timestamp}(Q)$, then the **read** operation is executed, and $R\text{-timestamp}(Q)$ is set to **max**($R\text{-timestamp}(Q)$, $TS(T_i)$).



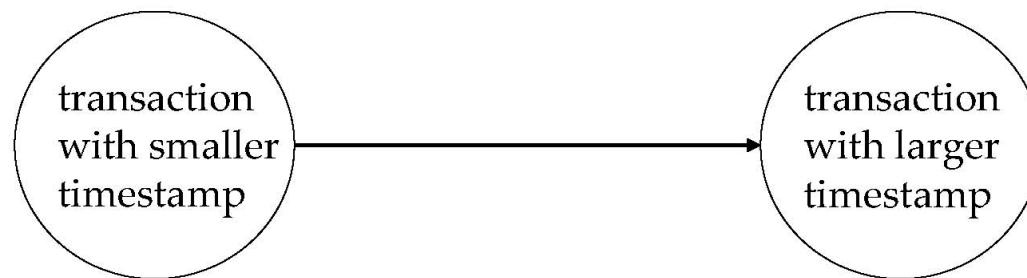
Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
 1. If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.
 - Hence, the **write** operation is rejected, and T_i is rolled back.
 2. If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q .
 - Hence, this **write** operation is rejected, and T_i is rolled back.
 - Otherwise, the **write** operation is executed, and $\text{W-timestamp}(Q)$ is set to $\text{TS}(T_i)$.



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

- Problem with timestamp-ordering protocol:
 - Suppose T_i aborts, but T_j has read a data item written by T_i
 - Then T_j must abort; if T_j had been allowed to commit earlier, the schedule is not recoverable.
 - Further, any transaction that has read a data item written by T_j must abort
 - This can lead to **cascading rollback** --- that is, a chain of rollbacks
- 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 $\text{TS}(T_i) < \text{W-timestamp}(Q)$, 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.

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

$\text{TS}(T_{27}) < \text{W-timestamp}(Q)$



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.
 - 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.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation



Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
 - $\text{Start}(T_i)$: the time when T_i started its execution
 - $\text{Validation}(T_i)$: the time when T_i entered its validation phase
 - $\text{Finish}(T_i)$: the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
 - Thus $\text{TS}(T_i)$ is given the value of $\text{Validation}(T_i)$.
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
 - because the serializability order is not pre-decided, and
 - relatively few transactions will have to be rolled back.



Validation Test for Transaction T_j

- If for all T_i with $\text{TS}(T_i) < \text{TS}(T_j)$ either one of the following condition holds:
 - $\text{finish}(T_i) < \text{start}(T_j)$
 - $\text{start}(T_j) < \text{finish}(T_i) < \text{validation}(T_j)$ 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:* Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
 - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
 - the writes of T_i do not affect reads of T_j since 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) $\langle validate \rangle$ display ($A + B$)	$\langle validate \rangle$ write (B) write (A)



Multiversion Schemes

- Multiversion schemes **keep old versions of data item** to increase concurrency.
 - Multiversion Timestamp Ordering
 - Multiversion Two-Phase Locking
- 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, and return the value of the selected version.
- **reads** never have to wait as **an appropriate version is returned immediately.**



Multiversion Timestamp Ordering

- Each data item Q has a sequence of versions $\langle Q_1, Q_2, \dots, Q_m \rangle$. 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
- when a transaction T_i creates a new version Q_k of Q , Q_k 's W-timestamp and R-timestamp are initialized to $TS(T_i)$.
- R-timestamp of Q_k is updated whenever a transaction T_j reads Q_k , and $TS(T_j) > R\text{-timestamp}(Q_k)$.



Multiversion Timestamp Ordering (Cont)

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



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



Insert and Delete Operations

- If two-phase locking is used :
 - A **delete** operation may be performed only if the transaction deleting the tuple has an **exclusive lock** on the tuple to be deleted.
 - A transaction that inserts a new tuple into the database is given an **X-mode lock on the tuple**
- Insertions and deletions can lead to the **phantom phenomenon**.
 - A transaction that scans a relation
 - ▶ (e.g., find sum of balances of all accounts in Perryridge)
 - and a transaction that inserts a tuple in the relation
 - ▶ (e.g., insert a new account at Perryridge)
 - (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 account, but reads some other tuple written by the update transaction



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 many database systems, **read committed** is the default consistency level
 - has to be explicitly changed to serializable when required
 - ▶ **set isolation level serializable**

```
set transaction isolation level serializable;
set transaction isolation level read committed;
```



Transactions across User Interaction

- Many applications need transaction support across user interactions
 - Can't use locking
 - Don't want to reserve database connection per user
- 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 where acctId = 23 and r.version = :version**
- Equivalent to **optimistic concurrency control without validating read set**
- Used internally in Hibernate ORM system, and manually in many applications



Chapter 11: Indexing and Storage



Textbook: Chapter 10 & 11



Chapter 11: Indexing and Storage

- File Organization
- Indexing
 - Basic Concepts
 - B+-Trees
 - Static Hashing
 - Index Definition in SQL



File Organization



File Organization

- The database is stored as a collection of **files**. Each file stores **records** (tuples from a table). A record is a sequence of **fields** (the attributes of a tuple).
- Reading one record of a time from disk would be very slow (random access)
 - Organize our database files in pages (size of block or larger)
 - Read/write data in units of pages
 - One page will usually contain several records
- One approach:
 - assume record size is fixed
 - each file has records of one particular type only
 - different files are used for different relations

This case is easiest to implement; will consider variable length records later.



Fixed-Length Records

■ Simple approach:

- Store record i starting from byte $n * (i - 1)$, where n is the size of each record. Put maximal P / n records on each page.
- Record access is simple but records may cross blocks
 - ▶ Modification: do not allow records to cross block boundaries

■ Deletion of record i : alternatives:

- move records $i + 1, \dots, n$ to $i, \dots, n - 1$
- move record n to i
- do not move records, but link all free records on a *free list*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Free Lists

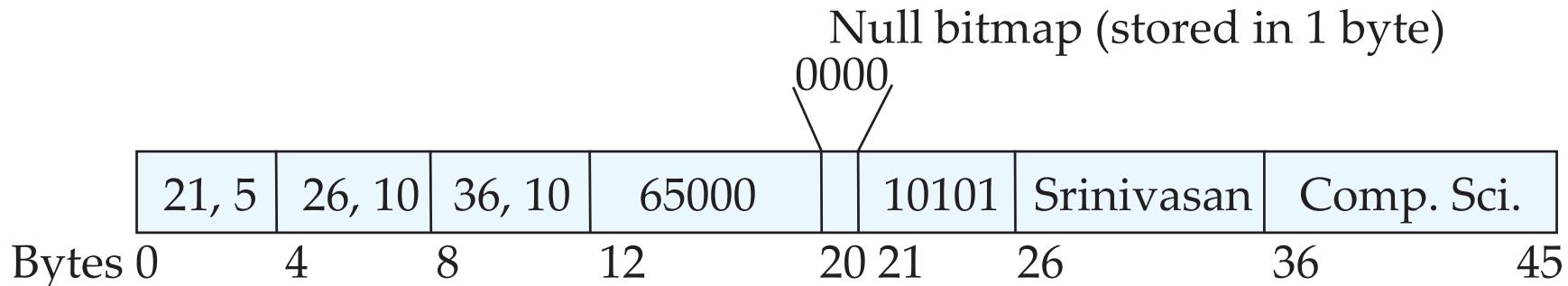
- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as **pointers** since they “point” to the location of a record.

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



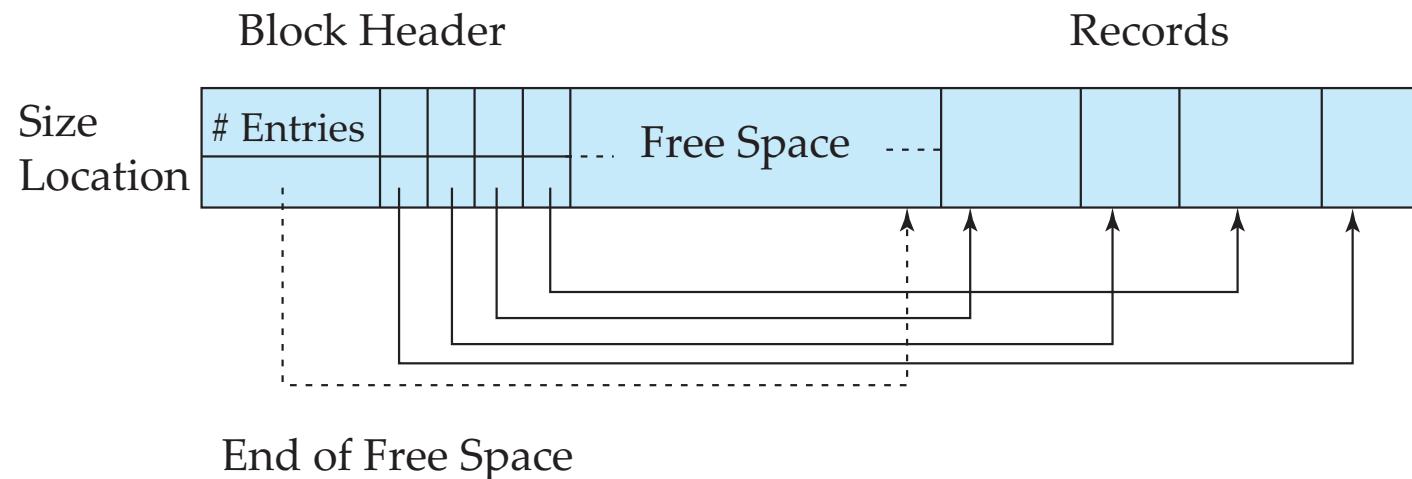
Variable-Length Records

- Variable-length records arise in database systems in several ways:
 - Storage of multiple record types in a file.
 - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
 - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap





Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
 - number of record entries
 - end of free space in the block
 - location and size of each record
- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to **the entry for the record in header**.



Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
 - Deletion efficient
 - Insertion efficient
 - Search is expensive
 - ▶ Example: Get instructor with name David
 - Have to search through all instructors
- **Sequential** – store records in sequential order, based on the value of some search key of each record
 - Deletion expensive and/or waste of space
 - Insertion expensive and/or waste of space
 - Search is efficient (e.g., binary search)
 - ▶ As long as the search is on the search key we are ordering on



Indexing and Hashing



Basic Concepts

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



- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in some 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
- Insertion time
- Deletion time
- Space overhead

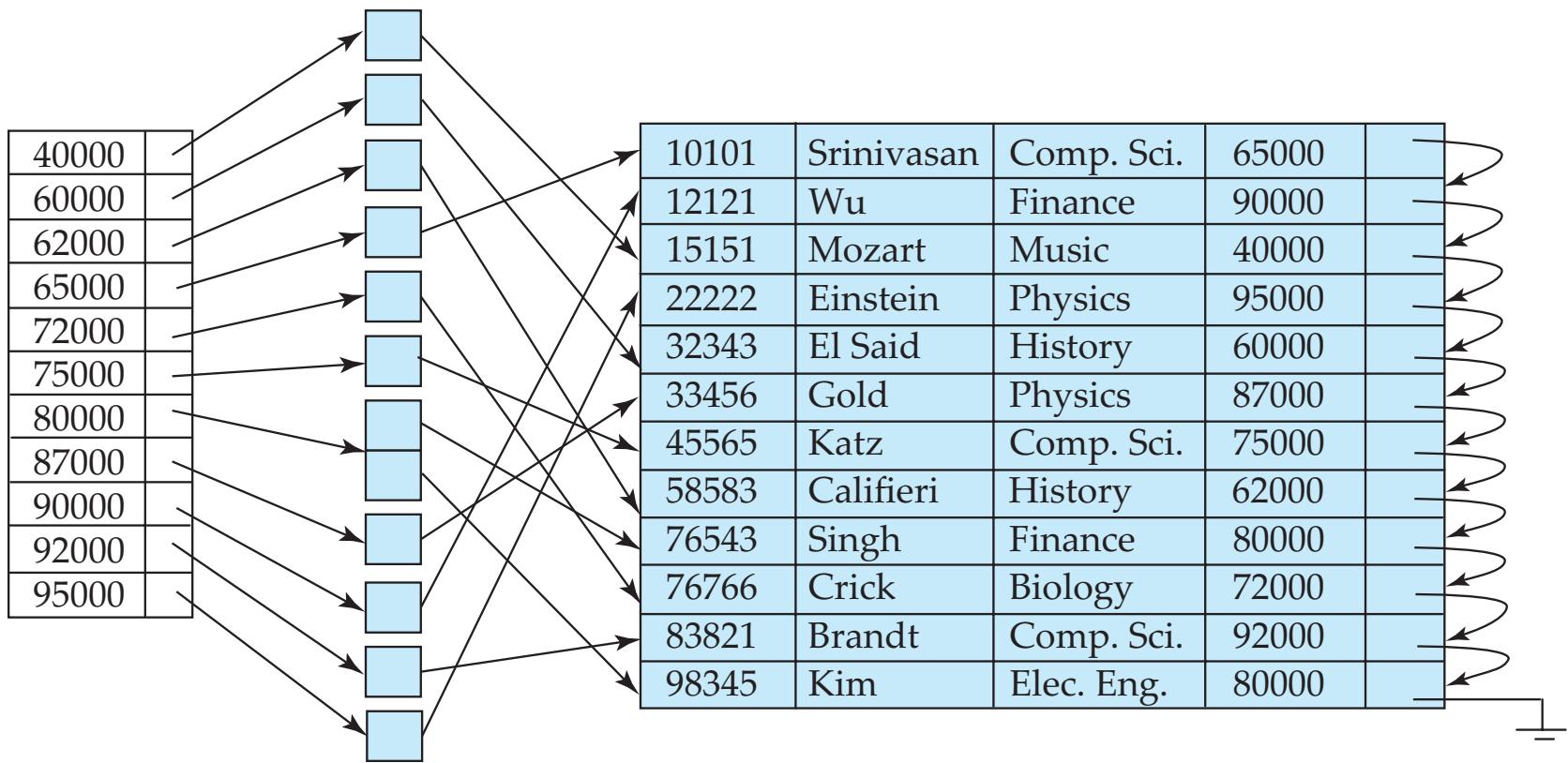


Ordered Indices

- In an **ordered index**, index entries are stored and 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.



Secondary Indices Example



- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be **dense**



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: **Updating** indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- 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
 - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

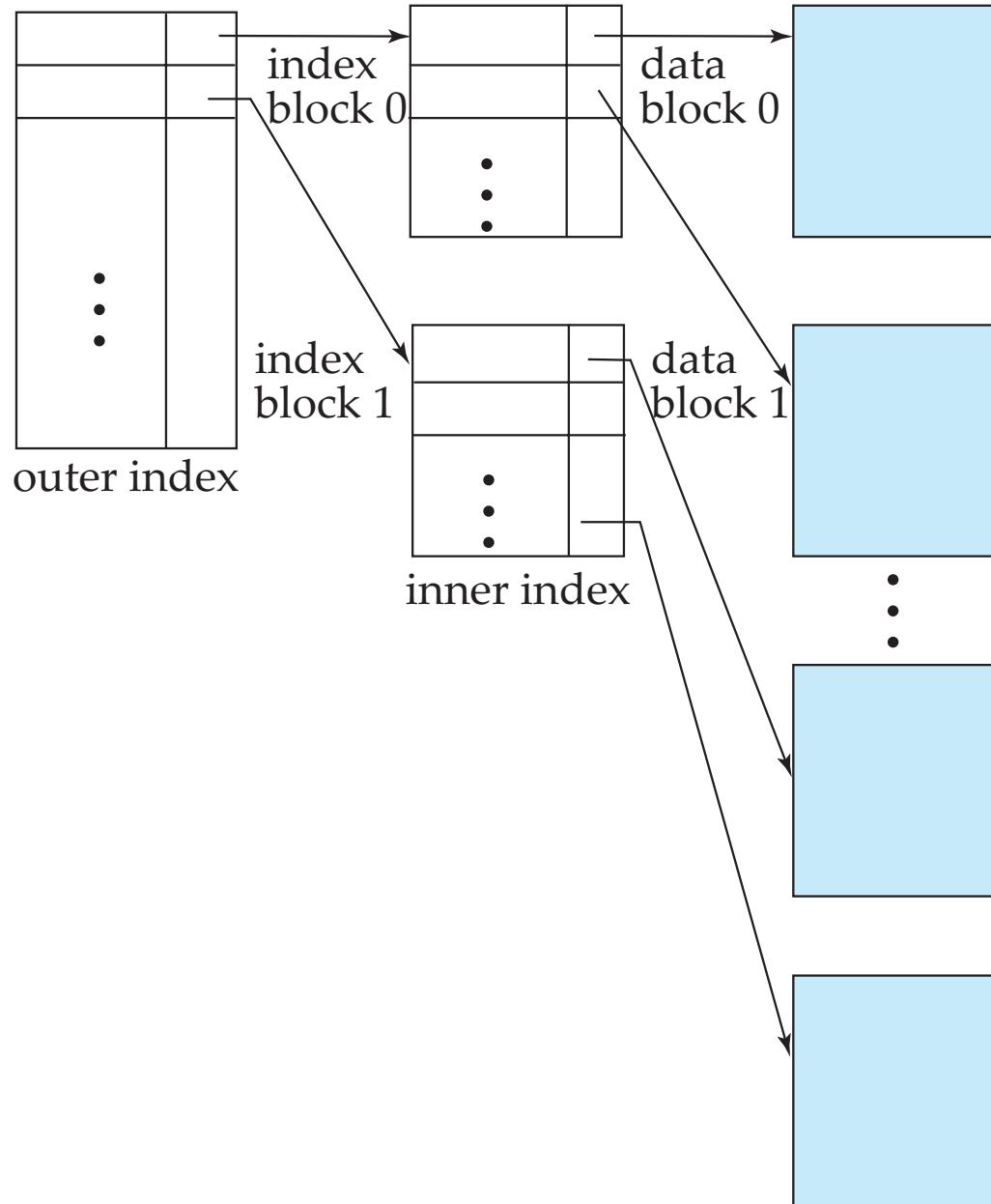


Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: 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.

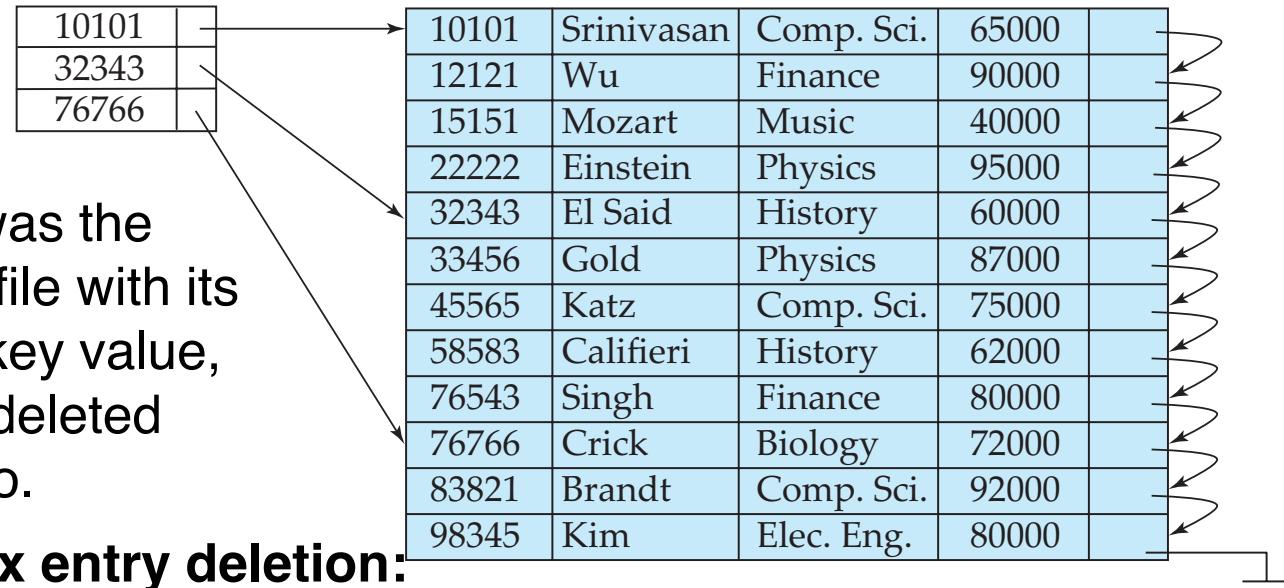


Multilevel Index (Cont.)





Index Update: Deletion



- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- Single-level index entry deletion:**

- **Dense indices** – deletion of search-key is similar to file record deletion.
- **Sparse indices** –
 - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
 - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



Index Update: Insertion

■ Single-level index insertion:

- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices** – if the search-key value does not appear in the index, insert it.
- **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
 - ▶ If a new block is created, the first search-key value appearing in the new block is inserted into the index.

■ Multilevel insertion and deletion: algorithms are simple extensions of the single-level algorithms



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 *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
 - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a **secondary index** with an index record for each search-key value



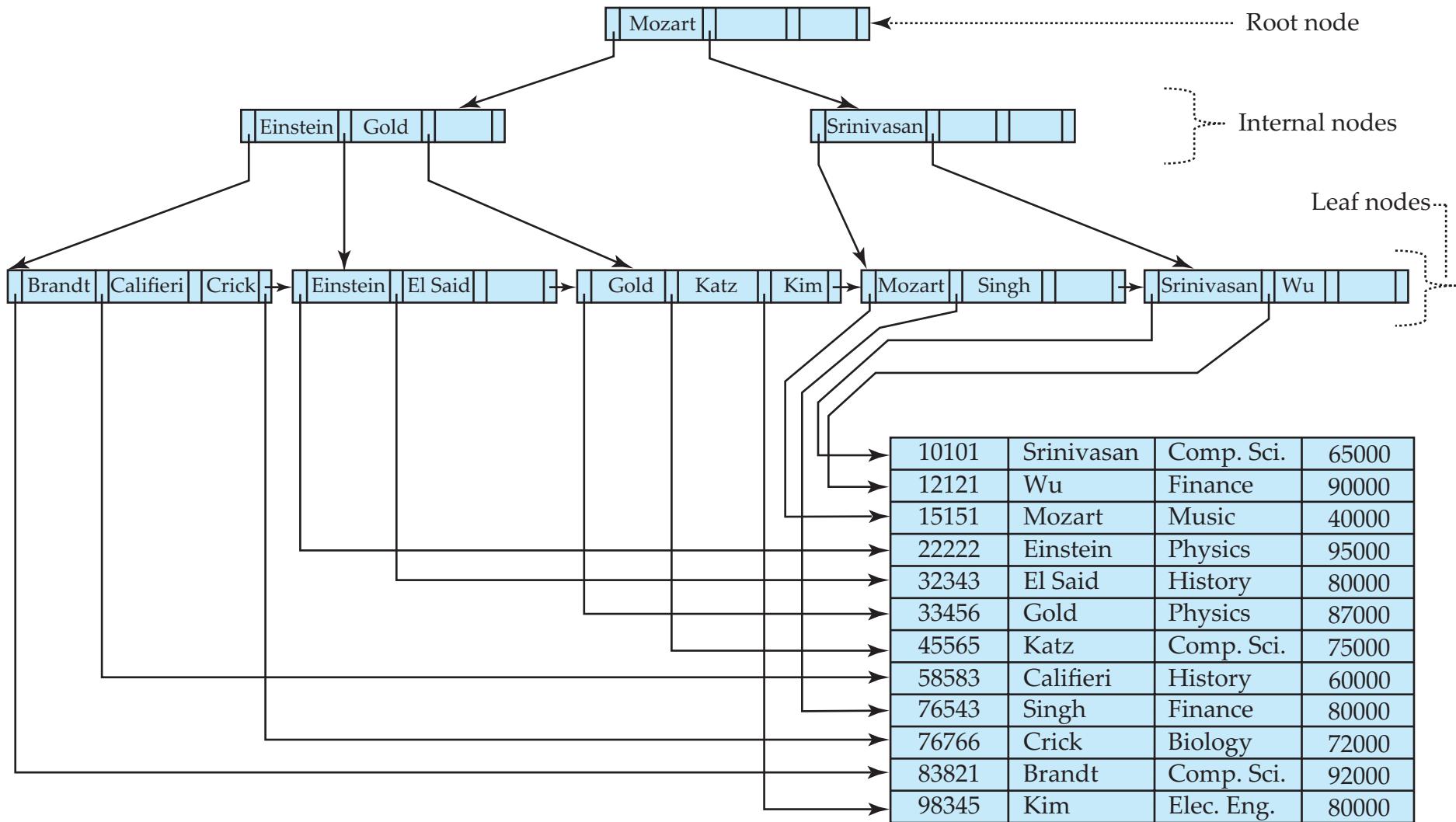
B⁺-Tree Index

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.
- (Minor) disadvantage of B⁺-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-trees outweigh disadvantages
 - B⁺-trees are used extensively



Example of B+-Tree





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
- n : number of pointers in each node
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- 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

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------

- 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

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

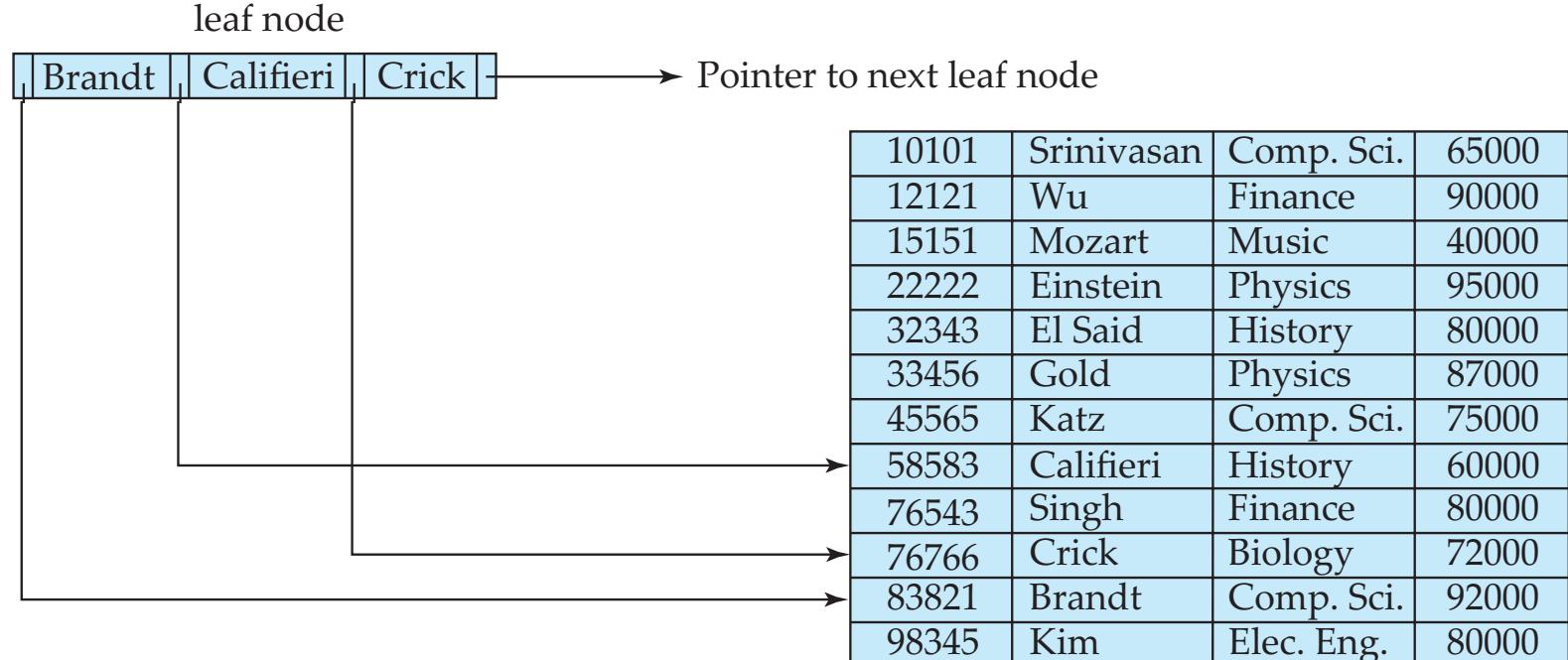
(Initially assume no duplicate keys, address duplicates later)



Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, pointer P_i points to a file record with search-key value K_i ,
- If L_i, L_j are leaf nodes and $i < j$, L_i 's search-key values are less than or equal to L_j 's search-key values
- P_n points to next leaf node in search-key order





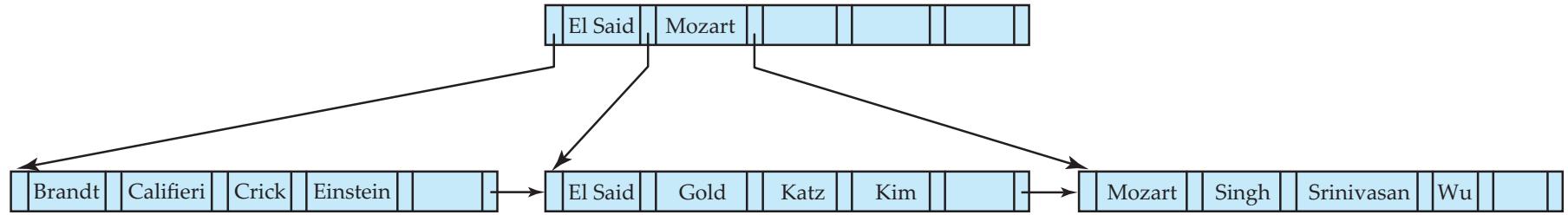
Non-Leaf Nodes in B⁺-Trees

- 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 $2 \leq i \leq n - 1$, 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_i
 - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

P_1	K_1	P_2	\dots	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	---------	-----------	-----------	-------



Example of B⁺-tree



B⁺-tree for *instructor* file ($n = 6$)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.



Observations about B+-trees

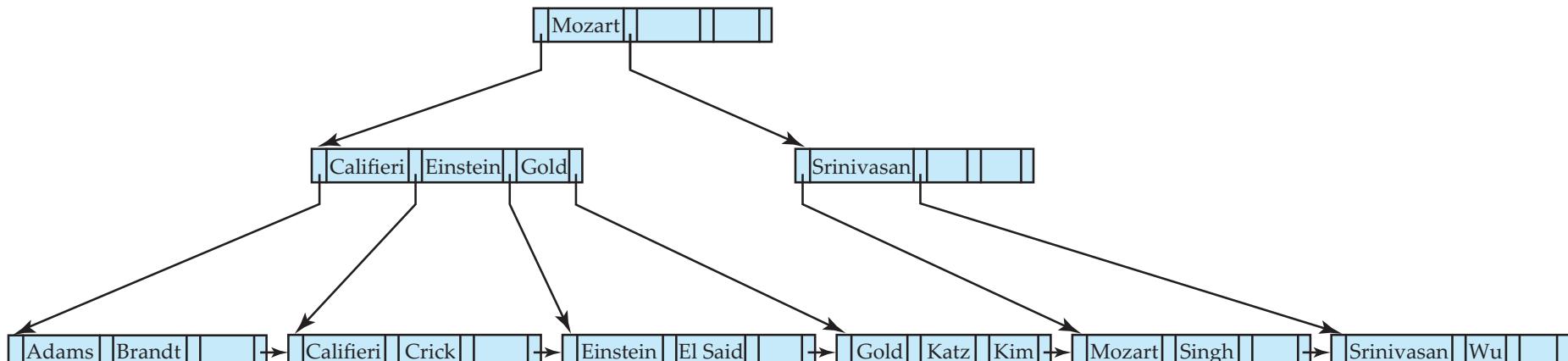
- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
 - ▶ Level below root has at least $2 * \lceil n/2 \rceil$ values
 - ▶ Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ values
 - ▶ .. etc.
 - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
 - 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 (as we shall see).



Queries on B⁺-Trees

- Find record with search-key value V .

1. $C = \text{root}$
2. While C is not a leaf node {
 1. Let i be least value s.t. $V \leq K_i$.
 2. If no such exists, set $C = \text{last non-null pointer in } C$
 3. Else { if ($V = K_i$) Set $C = P_{i+1}$ else set $C = P_i$ }
4. Let i be least value s.t. $K_i = V$
5. If there is such a value i , follow pointer P_i to the desired record.
6. Else no record with search-key value k exists.





Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more 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 tree 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



Updates on B+-Trees: Insertion

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



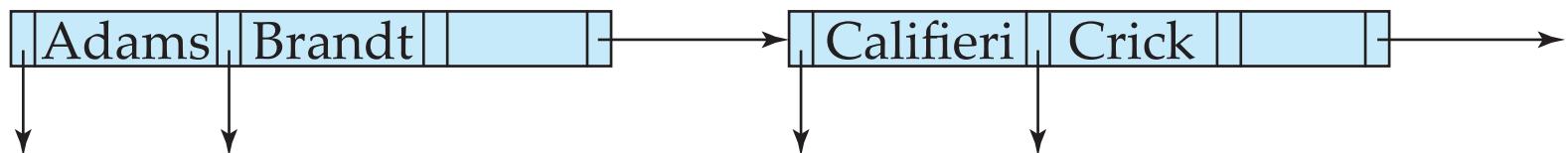
Updates on B⁺-Trees: Insertion (Cont.)

■ Splitting a leaf 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.

■ Splitting of nodes proceeds upwards till a node that is not full is found.

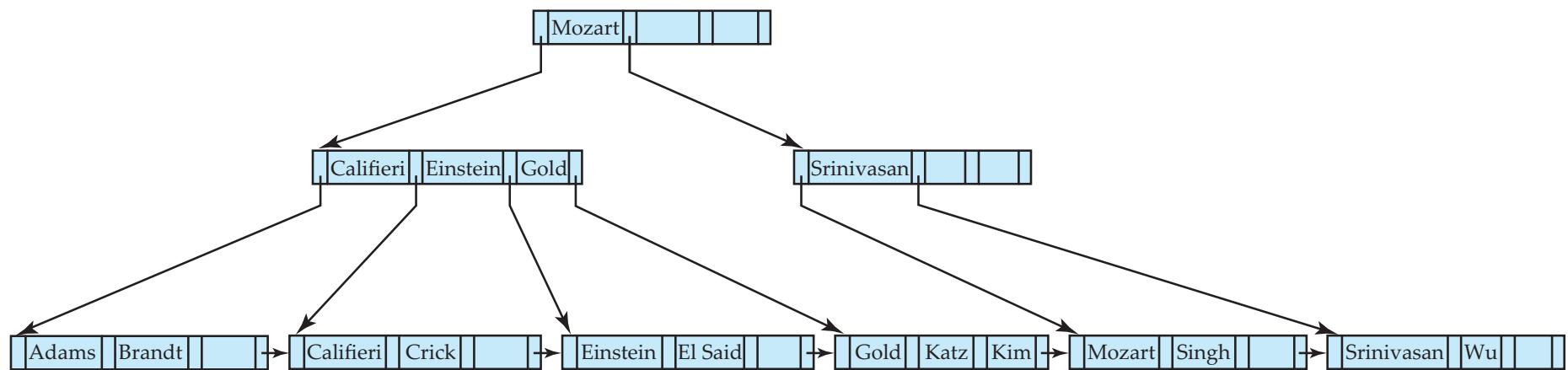
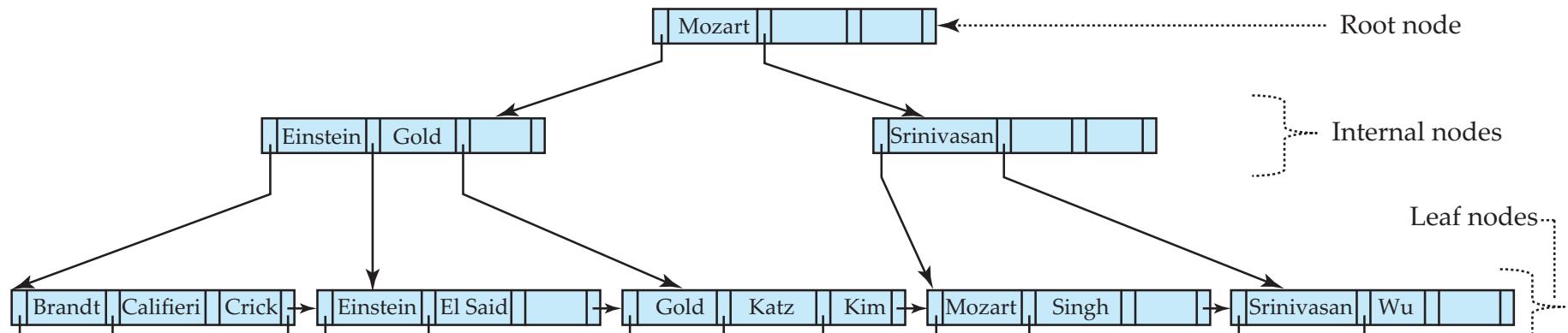
- In the worst case the root node may be split increasing the height of the tree by 1.



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent



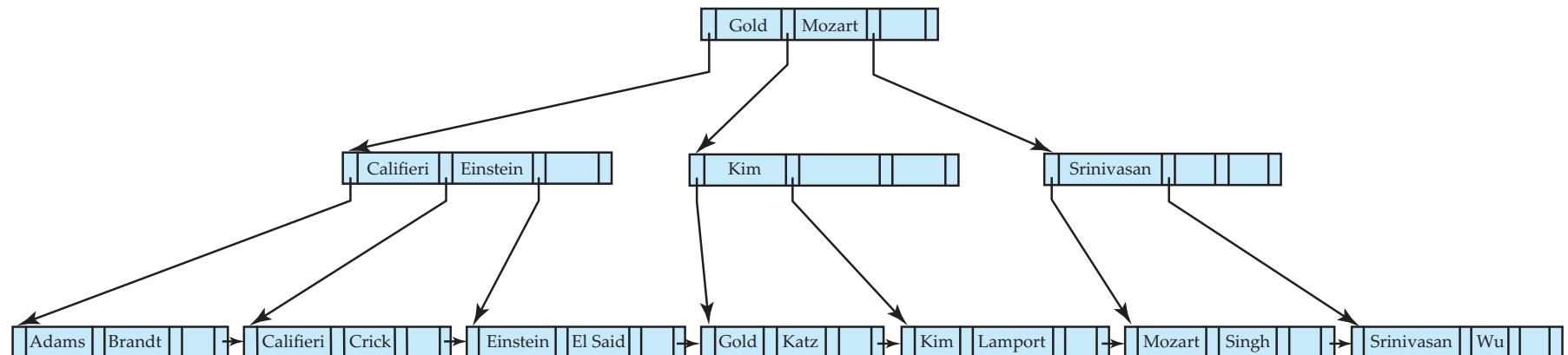
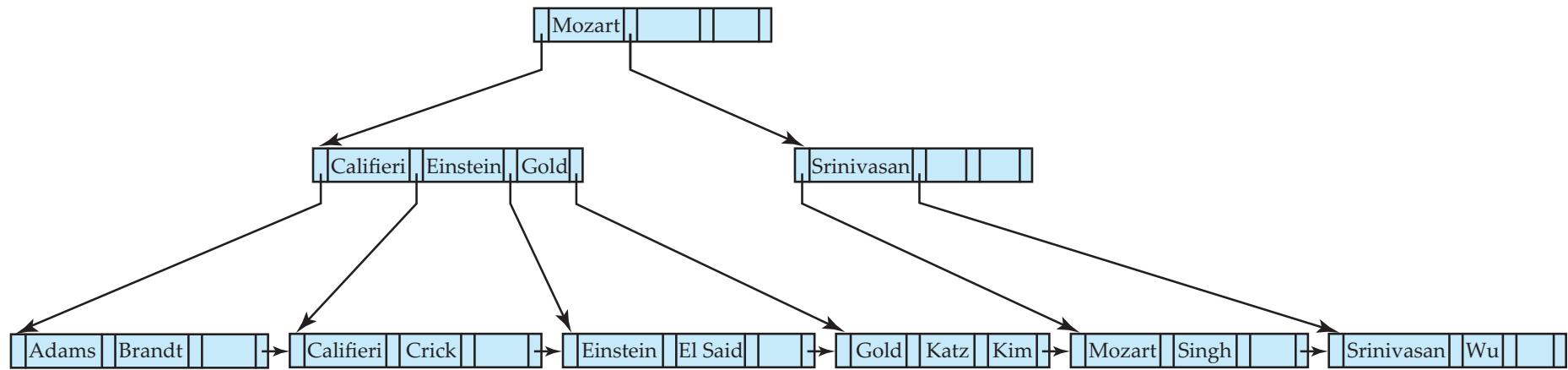
B⁺-Tree Insertion



B⁺-Tree before and after insertion of “Adams”



B⁺-Tree Insertion



B⁺-Tree before and after insertion of “Lamport”

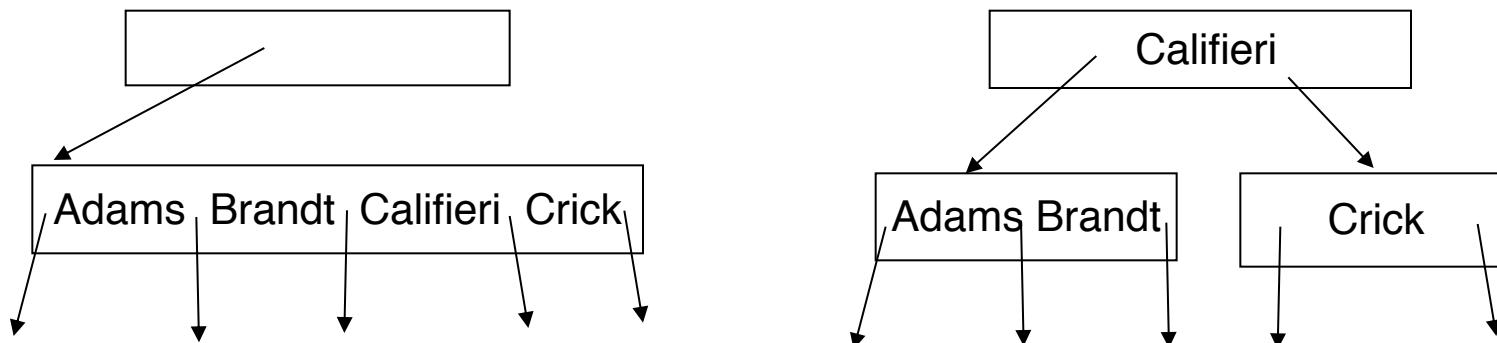


Insertion in B+-Trees (Cont.)

■ Splitting a non-leaf node: when inserting (k,p) into an already full internal node N

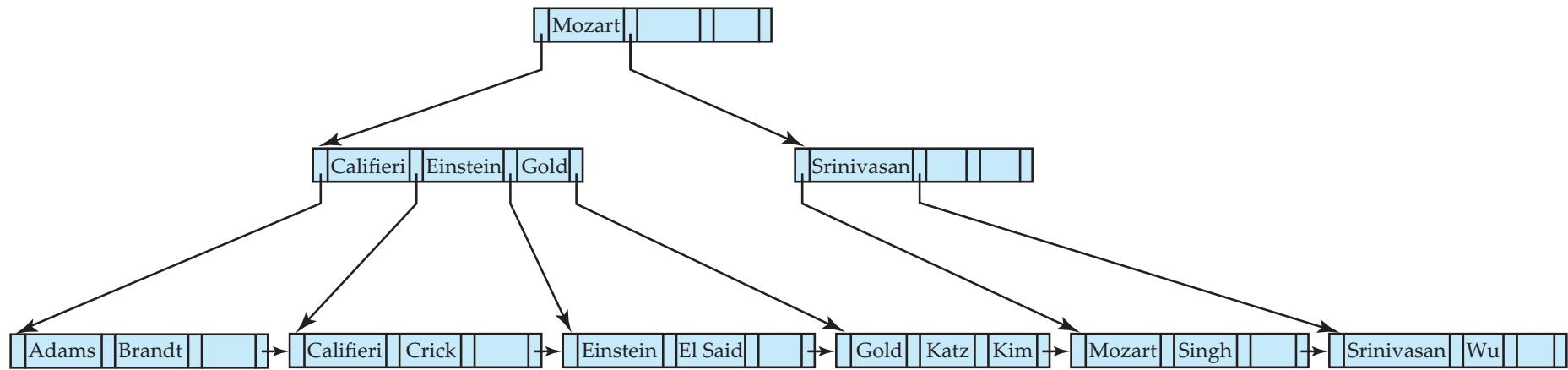
- Copy N to an in-memory area M with space for $n+1$ pointers and n keys
- Insert (k,p) into M
- Copy $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
- Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ from M into newly allocated node N'
- Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

■ Read pseudocode in book!

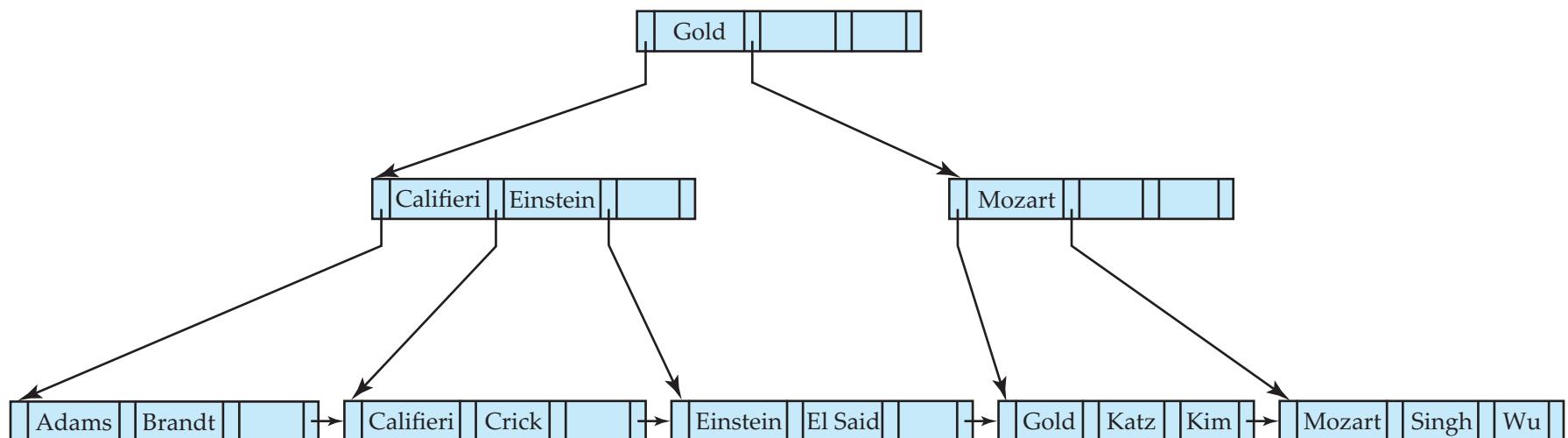




Examples of B⁺-Tree Deletion



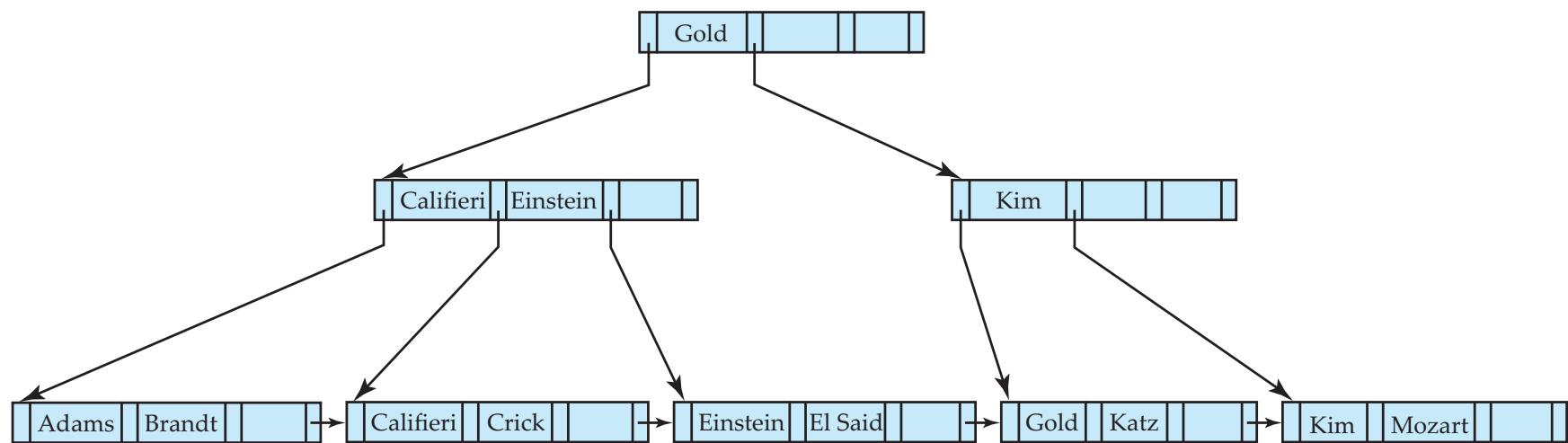
Before and after deleting “Srinivasan”



- Deleting “Srinivasan” causes merging of under-full leaves



Examples of B+-Tree Deletion (Cont.)

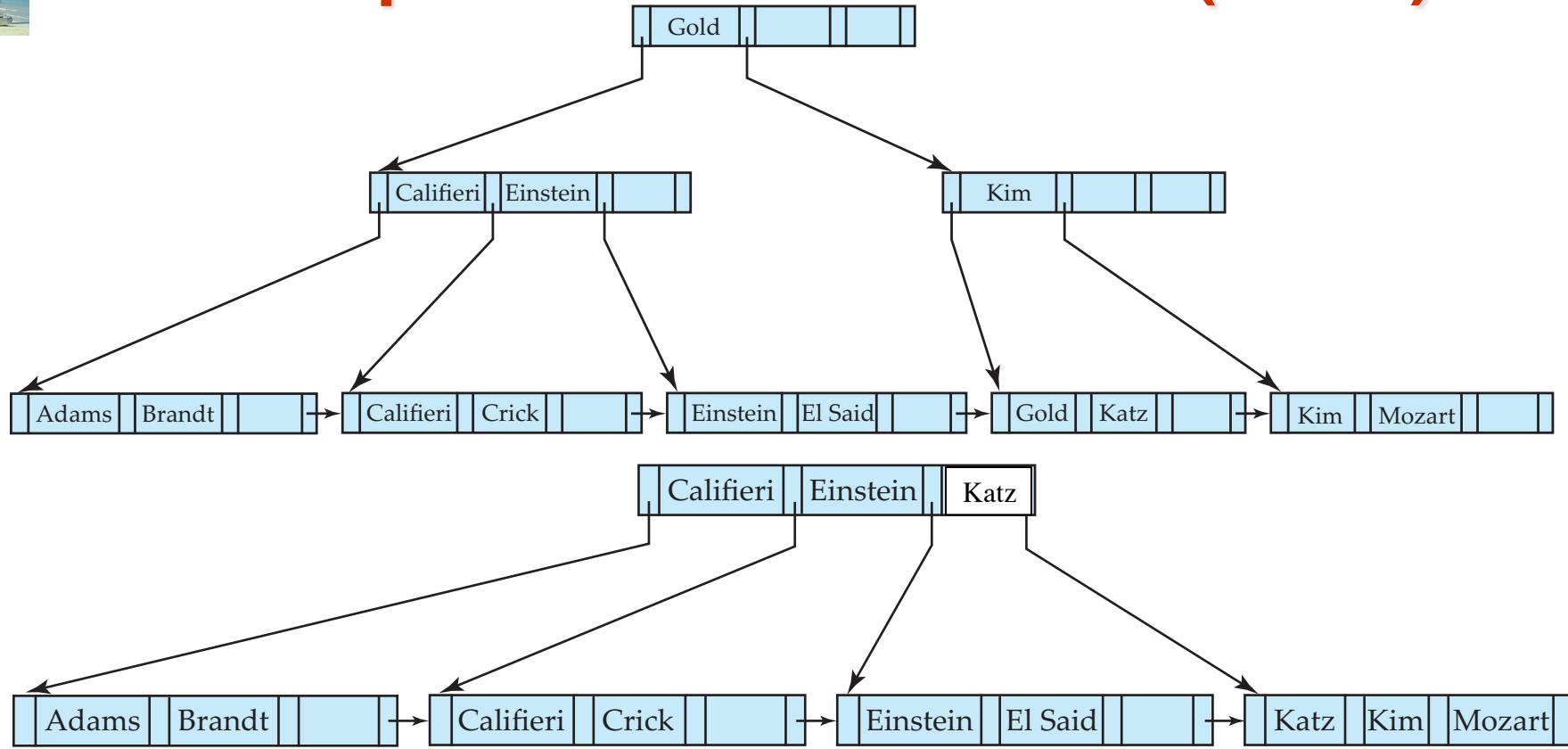


Deletion of “Singh” and “Wu” from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result



Example of B⁺-tree Deletion (Cont.)



Before and after deletion of “Gold” from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
 - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted



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 merge siblings:
 - 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, but the entries in the node and a *sibling* do not fit into a single node, then **redistribute pointers**:
 - 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.

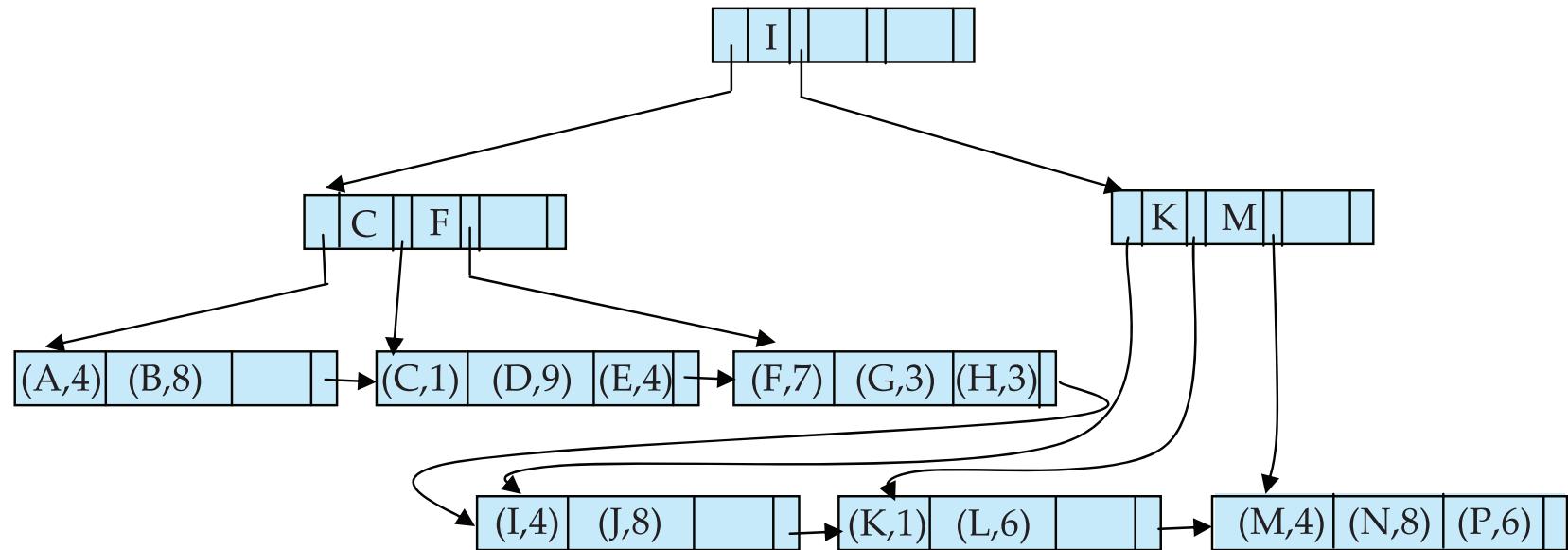


B⁺-Tree File Organization

- 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.
- Leaf nodes are still required to be half full
 - 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.
- 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



Hashing



Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically 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

Hash file organization of *instructor* file, using *dept_name* as key
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the i th character is assumed to be the integer i .
- The hash function returns the sum of the binary representations of the characters modulo 10
 - E.g. $h(\text{Music}) = 1 \quad h(\text{History}) = 2$
 $h(\text{Physics}) = 3 \quad h(\text{Elec. Eng.}) = 3$



Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key
(see previous slide for details).



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 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. .



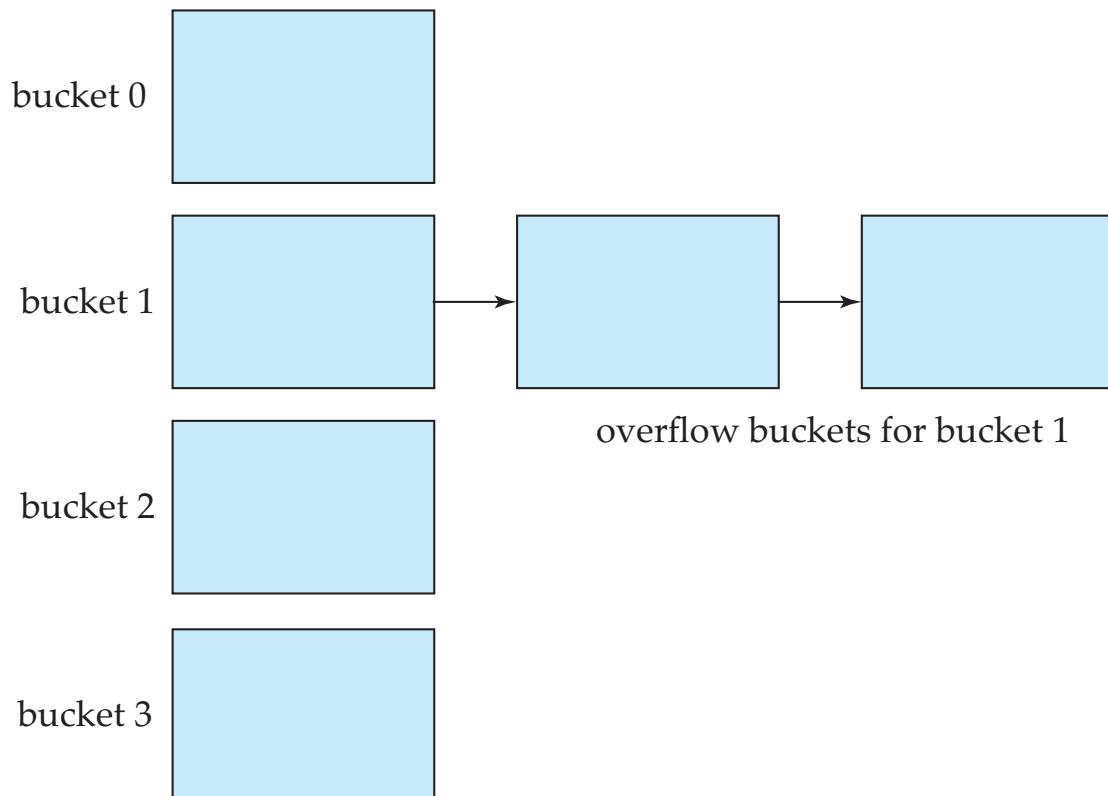
Handling of Bucket Overflows

- 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, 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.



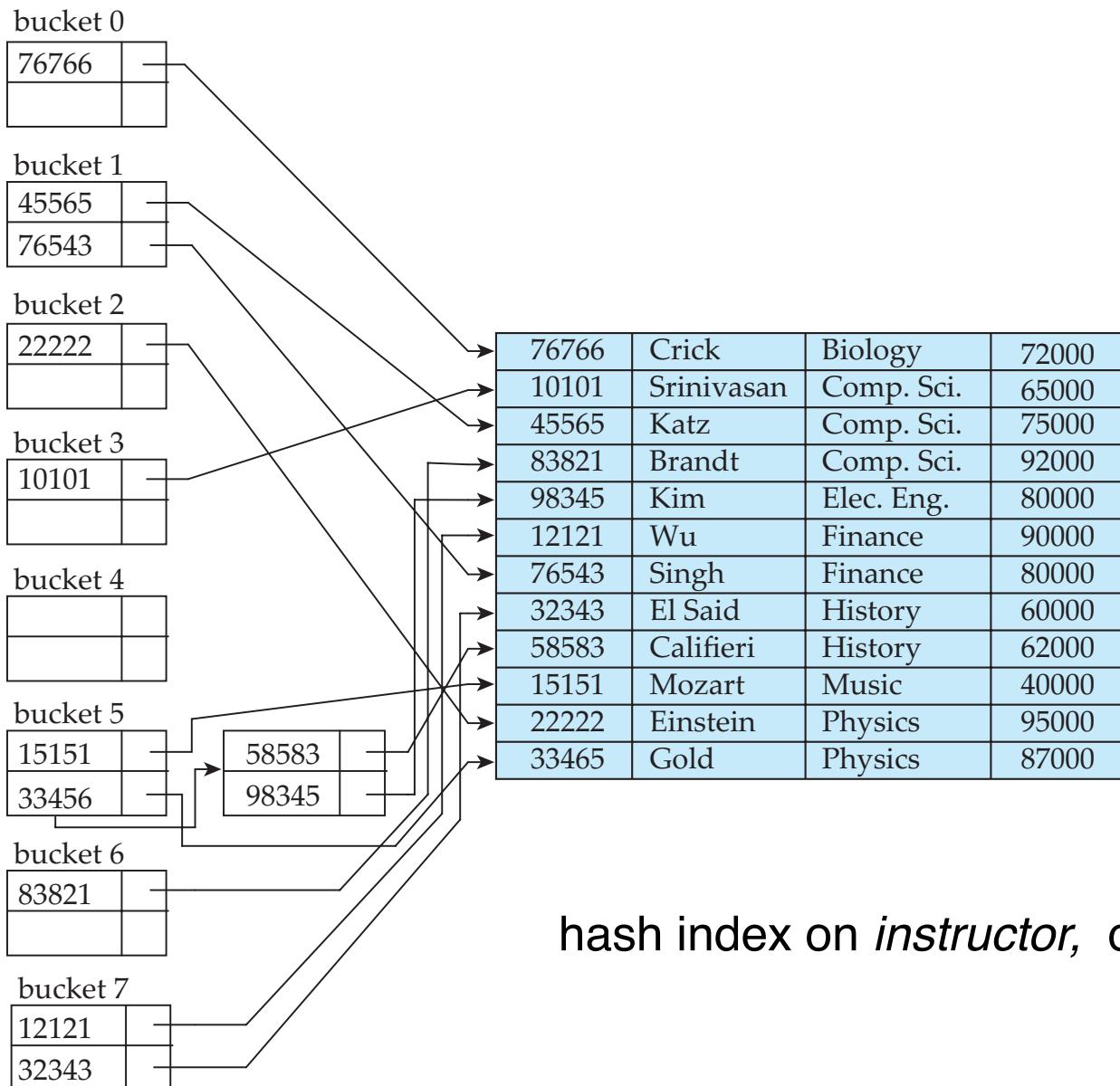


Hash Indices

- 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 indices (why?)**
 - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
 - However, we use the term hash index to refer to both secondary index structures and hash organized files.



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 or shrink with time.
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
 - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
 - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
 - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.



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>
```

- Most database systems allow specification of type of index, and clustering.



End of Chapter