

DATABASE MANAGEMENT SYSTEMS

UNIT V: Transaction Concept & Indexing Techniques

Syllabus:

Transaction Concept: Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability, Failure Classification, Storage, Recovery and Atomicity, Recovery algorithm.

Indexing Techniques: B+ Trees: Search, Insert, Delete algorithms, File Organization and Indexing, Cluster Indexes, Primary and Secondary Indexes , Index data Structures, Hash Based Indexing: Tree base Indexing ,Comparison of File Organizations, Indexes and Performance Tuning.

Introduction

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.
- The transaction consists of all operations executed between the statements **begin** and **end** of the transaction
- **Transaction operations:** Access to the database is accomplished in a transaction by the following two operations:

read (X): Performs the reading operation of data item X from the database

write (X): Performs the writing operation of data item X to the database

- A transaction must see a consistent database
- During transaction execution the database may be inconsistent
- When the transaction is *committed*, the database must be consistent
- Two main issues to deal with:

Failures, e.g. hardware failures and system crashes

Concurrency, for simultaneous execution of multiple transactions

ACID Properties

To preserve 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
- **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

Example of Fund Transfer: Let T_i be a transaction that transfers 50 from account A to B. This transaction can be illustrated as follows

Transfer \$50 from account A to B:

T_i : **read(A)**

$A := A - 50$

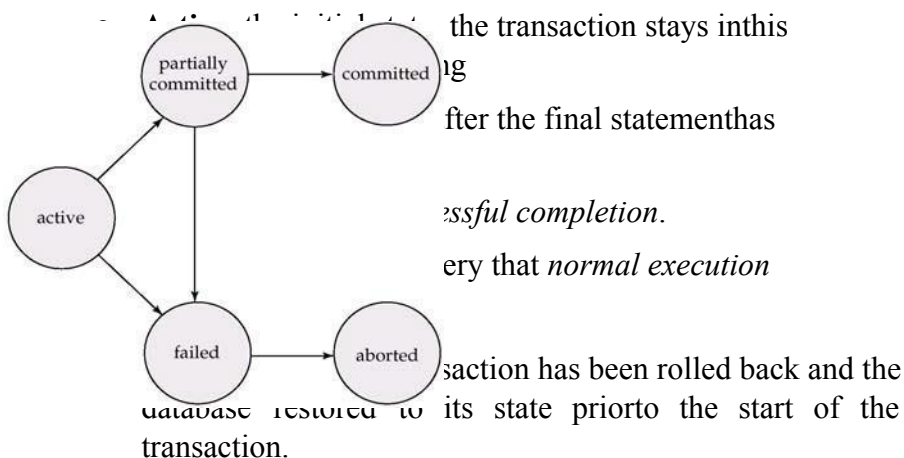
write(*A*)
read(*B*)
 $B := B + 50$
write(*B*)

- **Consistency:** the sum of *A* and *B* is unchanged by the execution of the transaction.
- **Atomicity:** if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Durability:** once the user has been notified that the transaction has completed, the updates to the database by the transaction must persist despite failures.
- **Isolation:** between steps 3 and 6, no other transaction should access the partially updated database, or else it will see an inconsistent state (the sum $A + B$ will be less than it should be).

Transaction and Schedules

- A transaction is seen by the DBMS as a series, or list of actions. We therefore establish a simple transaction model named as *transaction states*.

Transaction State: *A transaction must be one of the following states:*



Concurrent Execution and Schedules

Concurrent execution: executing transactions simultaneously has the following advantages:

- *increased processor and disk utilization, leading to better throughput*
- *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: these are mechanisms to achieve isolation

- *to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database*

Schedules: sequences that indicate 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

Example Schedules

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

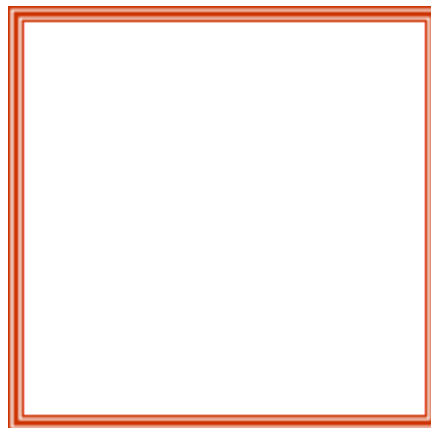
from A to B , and T_2 transfer 10% of the balance from A to B . The schedule (Schedule 1 in the text), in which T_1 is followed by T_2 .



Schedule 1

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

transactions defined previously. The following schedule is not a serial *uivalent* to above Schedule.



Schedule 2

- The following concurrent schedule does not preserve the value of the sum $A + B$

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



Schedule 3

Serializable Schedule

- A *serializable schedule* over a set S of committed transactions is a schedule whose effect on any consistent database is guaranteed to be identical to that of some complete serial schedule over S. i.e., even though the actions of transactions are interleaved, the result of executing transactions serially in different order may produce different results.
- **Example:** The schedule shown in the following figure is serializable.

T1	T2
R(A) W(A)	R(A) W(A)
R(B) W(B)	R(B) W(A)
Committ	Committ

Even though the actions of **T1** and **T2** are interleaved, the result of this schedule is equivalent to first running **T1** entirely and then running and **T2** entirely. Actually **T1**'s read and write of B is not influenced by **T2**'s actions on B, and the net effect is the same if these actions are the serial schedule First **T1**, then **T2**. This schedule is also serializable if first **T2**, then **T1**. Therefore if **T1** and **T2** are submitted concurrently to a DBMS, either of these two schedules could be chosen as first

- A DBMS might sometimes execute transactions which is not a serial execution i.e., not serializable.
- *This can be happen for two reasons:*
 - First the DBMS might use a concurrency control method that ensures the executed schedule itself.
 - Second, SQL gives programmers the authority to instruct the DBMS to choose non-serializable schedule.

Anomalies due to Interleaved execution

- There are three main situations when the actions of two transactions **T1** and **T2** conflict with each other in the interleaved execution on the same data object.
 - **Write-Read (WR) Conflict:** Reading Uncommitted data.
 - **Read-Write (RW) Conflict:** Unrepeatable Reads
 - **Write-Write (WW) Conflict:** Overwriting Uncommitted Data.

□ Reading Uncommitted Data (WR Conflicts)

- **Dirty Read:** The first source of anomalies is that a transaction **T2** could read a database object **A** that has been just modified by another transaction **T1**, which has not yet committed, such a read is called a *dirty read*.
- **Example:** Consider two transactions **T1** and **T2**, where **T1** stands for transferring \$100 from **A** to **B** and **T2** stands for incrementing both **A** and **B** by 6% of their accounts. Suppose that their actions are interleaved as follows:
 - (i) **T1** deducts \$100 from account **A**, then immediately
 - (ii) **T2** reads accounts of **A** and **B** adds 6% interest to each, and then,
 - (iii) **T1** adds \$100 to account **B**.

This corresponding schedule is illustrated as follows:

T1	T2
R(A) A: = A - 100W(A)	R(A) A: = A + 0.06 AW(A) R(B) B:= B+.06 BW(B) Commit
R(B) B: = B + 100W(B) Commit	

The problem here is **T2** has added incorrect 6% interest to each **A** and **B**. Because before commitment that \$100 is deducted from **A**, it has added 6% to account **A** before commitment that \$100 is credited to **B**, it has added 6% to account **B**. thus, the result of this schedule is different from the result of the other schedule which is serializable: first **T1** then **T2**.

□ Unrepeatable Reads (RW Conflicts)

- The second source of anomalies is that a transaction **T2** could change the value of an object **A** that has been read by a transaction **T1** and **T1** is still in progress. This situation causes a problem that, if **T1** tries to read the value of **A** again, it will get a different result, even though it has not modified **A** in the meantime. But, this situation could not arise in a serial execute of two transactions: this, it is called as *unrepeatable read*.
- **Example:** Suppose that both **T1** and **T2** reads the same value of **A**, say 5. Then **T1** has incremented **A** value to 6 but before commitment as **A** value 6, **T2** has decremented **A** value from 5 to 4. Thus, instead of answer of **A** value as 5, i.e., from 5 we got an answer 4 which is incorrect.

□ Overwriting Uncommitted Data (WW Conflicts)

- The third source of anomalies is that a transaction **T2** could overwrite the value of an object A, which has already been modified by a transaction **T1**, while **T1** is still in progress.
- **Example:** Suppose that A and B are two employees, and their salaries must be kept equal. Transaction **T1** sets their salaries to \$1000 and transaction **T2** sets their salaries to \$2000.

The following interleaving of the actions T1 and T2 occurs:

- i) **T1** sets A's salary to \$1000, at the same time, **T2** sets B's salary to \$2000.
- ii) **T1** sets B's salary is set to to \$2000, at the same time, **T2** sets A's salary to \$2000.

As a result A's salary is set to \$2000 and B's salary is set to \$1000, i.e., the result is not identical

- **Blind-Write:** Neither transaction reads a value before writing it-such a write is called a **blind-write**.

The above example is the best example of blind write because T1 and T2

are concentrating only on writing but not on reading.

Schedules involving aborted Transactions

- All transactions of aborted transactions are to be undone, and we can therefore imagine that they were never carried out to begin with.
- **Example:** Suppose that transaction **T1** deducts \$100 from account A then immediately before committing A's new value the transaction **T2** reads the current values of accounts A and B and adds 6% interest to each, then commits, but incidentally **T1** is aborted. So, we get incorrect result of transaction **T2** because **T1** was aborted in the middle of the process and **T2** has taken incorrect value of A by **T1** and added 6%. We say that such a schedule is **Unrecoverable Schedule**. The corresponding schedule is shown as follows:

T1	T2
R(A) A: = A - 100W(A)	R(A) A: = A + 0.06 AW(A) R(B) B:= B+.06 BW(B) Commit
Abort	

- Whereas, a recoverable schedule is one in which transactions read only the changes of committed transactions.

Serializability

Basic Assumption – Each transaction, on its own, preserves database consistency

- *i.e. serial execution 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 **conflict serializability** and **view serializability**

serializability and **view serializability**

Simplifying assumptions:

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

Conflict Serializability

Instructions li and lj of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both li and lj , and at least one of these instructions wrote Q .

1. $li = \text{read}(Q)$, $lj = \text{read}(Q)$. li and lj don't conflict.
2. $li = \text{read}(Q)$, $lj = \text{write}(Q)$. They conflict.
3. $li = \text{write}(Q)$, $lj = \text{read}(Q)$. They conflict
4. $li = \text{write}(Q)$, $lj = \text{write}(Q)$. They conflict

Intuitively, a conflict between li and lj forces a (logical) temporal order between them

If li and lj are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the ordering

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

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 $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.

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

The 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1 , by series of swaps of



non-conflicting instructions.

Therefore Schedule 3 is conflict serializable.

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, where Q is a data item and T_i is a transaction:

1. If T_i reads the initial value of Q in schedule S , then T_i must, in schedule S' , also read the initial value of Q
2. If T_i executes $\text{read}(Q)$ in schedule S , and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j
3. The transaction (if any) that performs the final $\text{write}(Q)$ operation in schedule S (for any data item Q) must perform the final $\text{write}(Q)$ operation in schedule S'


NB: View equivalence is also based purely on **reads** and **writes**

A schedule S is *view serializable* if it is view equivalent to a serial schedule

Every conflict serializable schedule is also view serializable

Schedule 9 (from book) — a schedule which is view-serializable but *not* conflict serializable

T_3	T_4	T_6
$\text{read}(Q)$		
	$\text{write}(Q)$	
$\text{write}(Q)$		
		$\text{write}(Q)$



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

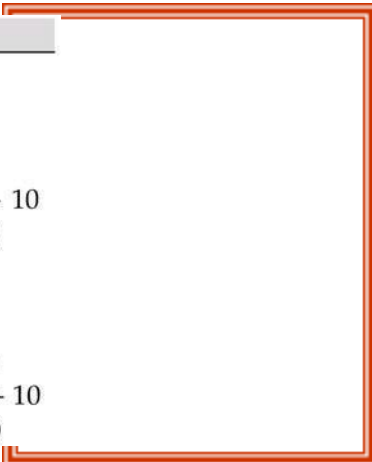
Other Notions of Serializability

This schedule produces the same outcome as the serial schedule $\langle T_1, T_5 \rangle$

However it is not conflict equivalent or view equivalent to it

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

T_1	T_5
$\text{read}(A)$	
$A := A - 50$	
$\text{write}(A)$	
	$\text{read}(B)$
	$B := B - 10$
	$\text{write}(B)$
$\text{read}(B)$	
$B := B + 50$	
$\text{write}(B)$	
	$\text{read}(A)$
	$A := A + 10$
	$\text{write}(A)$



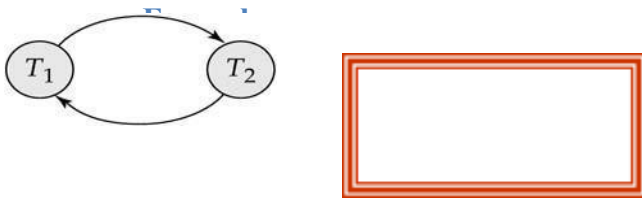
Testing for Serializability

Consider some schedule of a set of transactions T_1, T_2, \dots, T_n

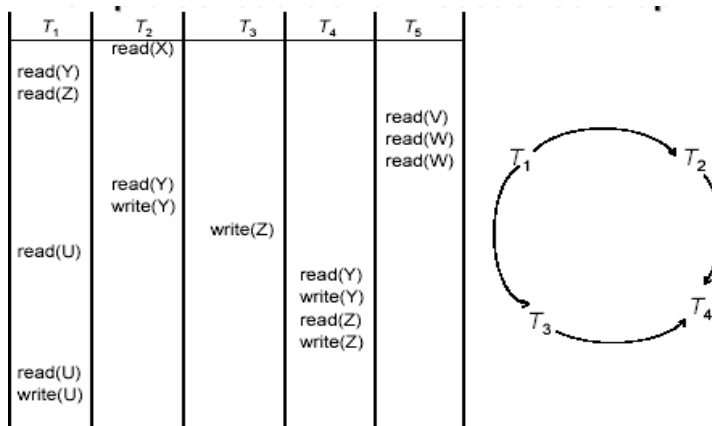
Precedence graph: a directed graph where the vertices are transaction names

We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item before T_j

We may label the arc by the item that was accessed



Example Schedule and Precedence Graph

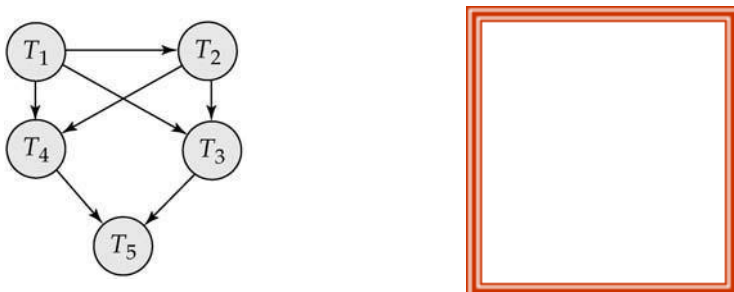


A schedule is **conflict serializable** if and only if its precedence graph is acyclic

- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph
- 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 this graph is $T_2 \prec T_1$
 $\square T_3 \prec T_4 \prec T_5$

The precedence graph test for conflict serializability must be modified to apply to a test for **view serializability**

- The problem of checking if a schedule is view serializable is *NP*-complete. Thus existence of an efficient algorithm is unlikely. However practical algorithms that just check some *sufficient conditions* for view serializability can still be used



Example of an acyclic precedence graph

Concurrency Control vs. Serializability Tests

Goal – to develop concurrency control protocols that will ensure serializability

These protocols will impose a discipline that avoids nonserializable schedules


A common concurrency control protocol uses *locks*

- while one transaction is accessing a data item, no other transaction can modify it
- require a transaction to lock the item before accessing it
- two standard lock modes are “shared” (read-only) and “exclusive” (read-write)

Recoverability


- 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 , the commit operation of T_i appears before the commit operation of T_j
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
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 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)
- 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

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



- *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

Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.

E.g. the **shadow-database** scheme:

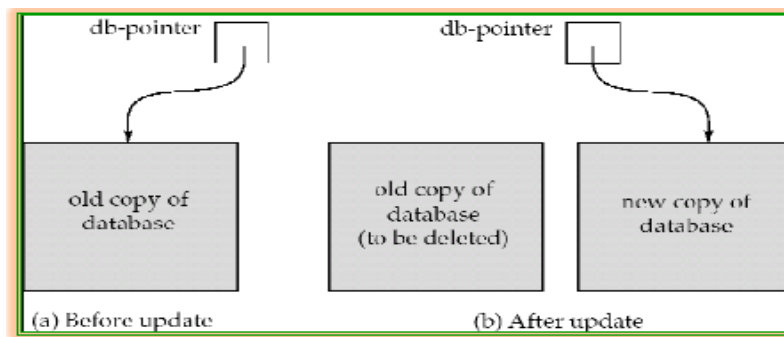
all updates are made on a *shadow copy* of the database

db_pointer is made to point to the updated shadow copy after the transaction reaches partial commit and all updated pages have been flushed to disk. **db_pointer** always points to the current consistent copy of the database.

In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

The shadow-database scheme:

Assumes that only one transaction is active at a time. Assumes disks do not fail Does not handle concurrent transactions



Storage Structure

- **Volatile storage:**
 - does not survive system crashes
 - examples: main memory, cache memory
- **Nonvolatile storage:**
 - survives system crashes
 - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
- **Stable storage:**
 - a mythical form of storage that survives all failures
 - approximated by maintaining multiple copies on distinct nonvolatile media

Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
 - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
 - Successful completion
 - Partial failure: destination block has incorrect information
 - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
 - Execute output operation as follows (assuming two copies of each block):
 1. Write the information onto the first physical block.
 2. When the first write successfully completes, write the same information onto the second physical block.
 3. The output is completed only after the second write successfully completes.

File Organization

A **file** is organized logically as a sequence of records. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*.

Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer.

Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created.

Two possible approaches to store records: • Record size is fixed • Record size is variable

Fixed-Length Records

As an example, let us consider a file of *instructor* records for our university database. Each record of this file is defined (in pseudocode) as:

```
type instructor = record
    ID varchar (5);
    name varchar(20);
    dept_name varchar (20);
    salary numeric (8,2);
end
```

We allocate the maximum number of bytes that each attribute can hold. Then, the instructor record is 53 bytes long.

A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on as shown below figure.

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

There are two problems with this simple approach:

1. Unless the block size happens to be a multiple of 53 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.
2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

To avoid the second problem, When a record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead shown in below figure.

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
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

Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record shown in below figure.

It is undesirable to move records to occupy the space freed by a deleted record, since doing so

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
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

requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.

A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

At the beginning of the file, we allocate a certain number of bytes as a **file header**. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted.

We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list**.

Below figure shows with the free list, after records 1, 4, and 6 have been deleted. On insertion of a new record, we use the 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.
- Record types that allow repeating fields, such as arrays or multisets.

Different techniques for implementing variable-length records exist. Two different problems must be solved by any such technique:

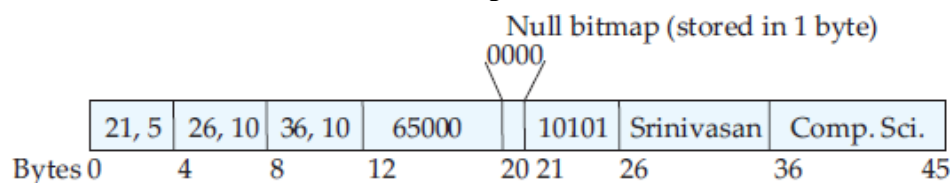
- How to represent a single record in such a way that individual attributes can be extracted easily.
- How to store variable-length records within a block, such that records in a block can be extracted easily.

Representation of variable-length record

The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes. Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value. Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (*offset*, *length*), where *offset* denotes where the data for that attribute begins within the record, and *length* is the length in bytes of the variable-sized attribute. The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length.

An example of such a record representation is shown in below figure. The figure shows an *instructor* record, whose first three attributes *ID*, *name*, and *dept name* are variable-length strings, and whose fourth attribute *salary* is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The *salary* attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

The figure also illustrates the use of a **null bitmap**, which indicates which attributes of the record



have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored.

Storing variable-length records in a block

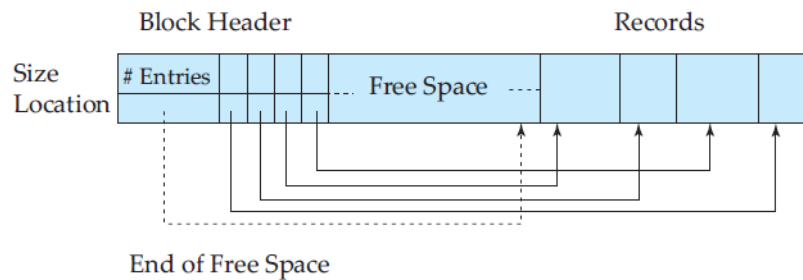
The **slotted-page structure** is commonly used for organizing records within a block. There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header.
2. The end of free space in the block.
3. An array whose entries contain the location and size of each record.

The actual records are allocated *contiguously* in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header array,

and the first record. If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* (its size is set to -1 , for example). Further, the records in the block before the deleted record are moved, so that the free space created by the deletion gets occupied, and all free space is again between the final entry in the header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well. Records can be grown or shrunk by similar techniques, as long as there is space in the block. The cost of moving the records is not too high, since the size of a block is limited: typical values are around 4 to 8 kilobytes.



Organization of Records in Files

A relation is a set of records. Given a set of records, the next question is how to organize them in a file. Several of the possible ways of organizing records in files are:

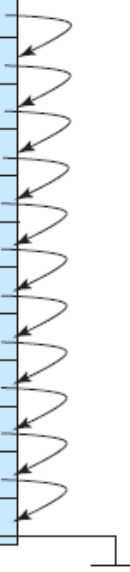
- **Heap file organization.** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.
- **Sequential file organization.** Records are stored in sequential order, according to the value of a “search key” of each record.
- **Hashing file organization.** A hash function is computed on some attribute of each record. The result of the hash function specifies in which block of the

Sequential File Organization

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. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

Blow figure shows a sequential file of *instructor* records taken from our university example. In that example, the records are stored in search-key order, using *ID* as the search key.

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	



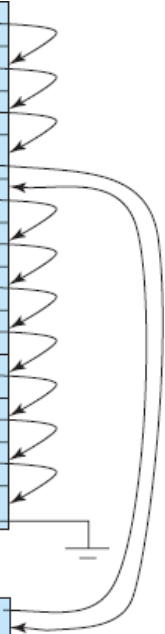
It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously. For insertion, we apply the following rules:

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.

Below figure shows the record after the insertion of the record (32222, Verdi, Music, 48000).

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	
-------	-------	-------	-------	--



Introduction to Indexing

An index for a file in a database system works in much the same way as the index in this textbook. If we want to learn about a particular topic (specified by a word or a phrase) in this textbook, we can search for the topic in the index at the back of the book, find the pages where it occurs, and then read the pages to find the information for which we are looking.

Database-system indices play the same role as book indices in libraries. For example, to retrieve a student record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate student record.

There are two basic kinds of indices:

- **Ordered indices.** Based on a sorted ordering of the values.
- **Hash indices.** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a *hash function*.

Index Evaluation Metrics

- **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
- **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.

An attribute or set of attributes used to look up records in a file is called a **search key**.

Ordered Indices

To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it.

A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a **clustering index** is an index whose search key also defines the sequential order of the file.

Clustering indices are also called **primary indices**; the term primary index may appear to denote an index on a primary key, but such indices can in fact be built on any search key.

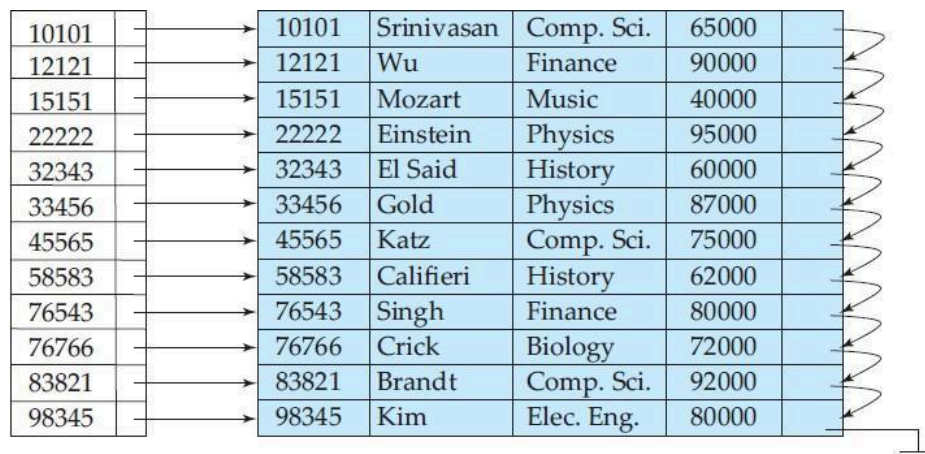
Indices whose search key specifies an order different from the sequential order of the file are called **nonclustering indices**, or **secondary indices**.

Dense and Sparse Indices

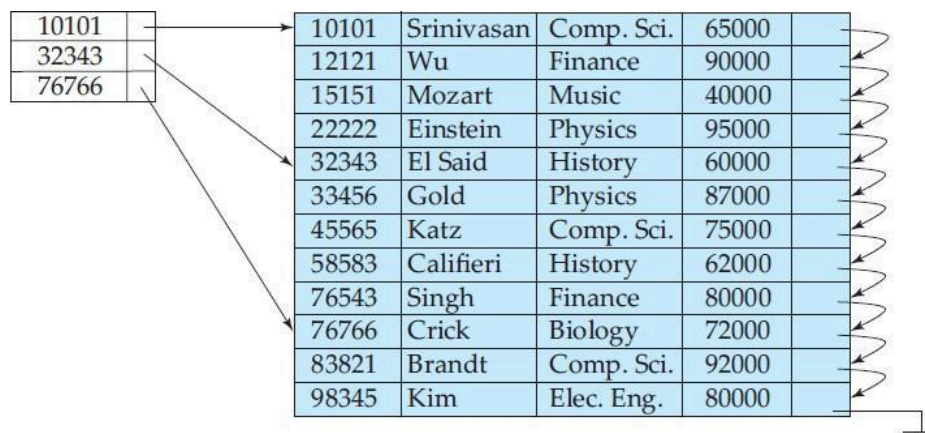
An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

There are two types of ordered indices that we can use:

- **Dense index:** In a dense index, an index entry appears for every search-key value in the file. In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.
- **Sparse index:** In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key, that is, if the index is a clustering index. As is true in dense indices, each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.



Dense index

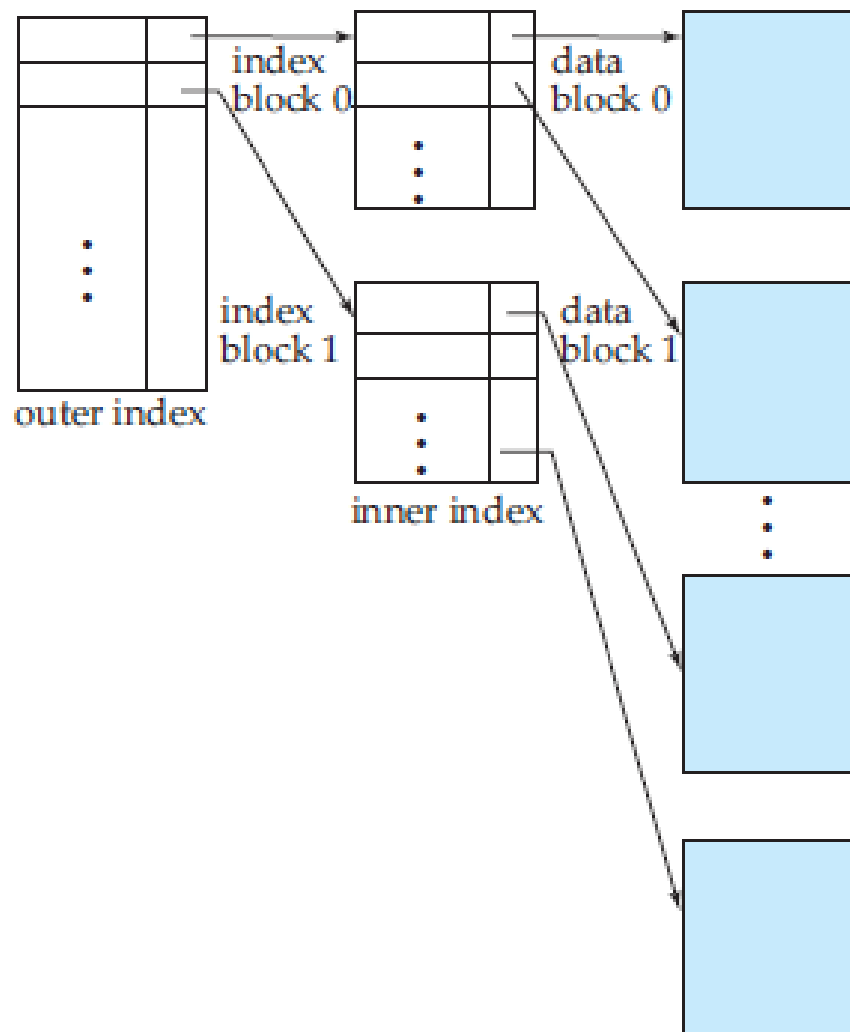


Sparse index

Multilevel Indices

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. (Even if an index is smaller than the main memory of a computer, main memory is also required for a number of other tasks, so it may not be possible to keep the entire index in memory.) The search for an entry in the index then requires several disk-block reads.

We treat the index just as we would treat any other sequential file, and construct a sparse outer index on the original index, which we now call the inner index, as shown in below figure. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire. The pointer points to a block of the inner index. We scan this block until we find the record that has the largest search-key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking.



Two level sparse index

Index Update

Regardless of what form of index is used, every index must be updated whenever a record is either inserted into or deleted from the file.

We first describe algorithms for updating single-level indices.

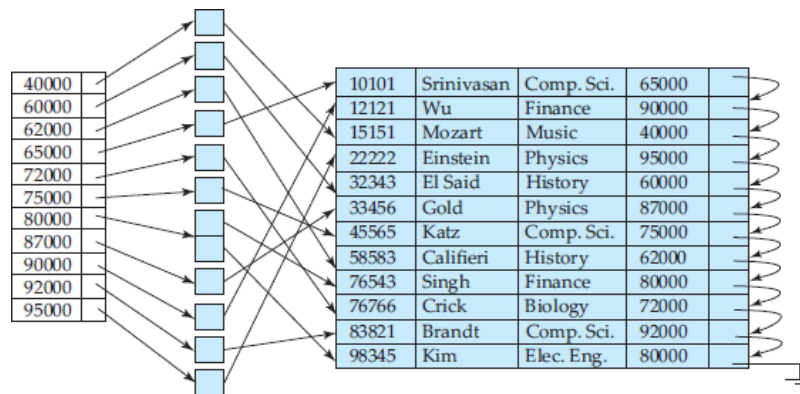
- **Insertion.** First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse:
 - **Dense indices:**
 1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
 2. Otherwise the following actions are taken:
 - a. If the index entry stores pointers to all records with the same search key value, the system adds a pointer to the new record in the index entry.
 - b. Otherwise, the index entry 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.
- **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 entry from the index.
 2. Otherwise the following actions are taken:
 - a. If the index entry stores pointers to all records with the same search key value, the system deletes the pointer to the deleted record from the index entry.
 - b. Otherwise, the index entry 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 entry to point to the next record.
 - **Sparse indices:**
 1. If the index does not contain an index entry 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 entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.

Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

Below figure shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.



B⁺ Tree Index Files

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data.

The **B⁺ tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.

A B⁺ tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each non leaf node in the tree has between $n/2$ and n children, where n is fixed for a particular tree.

A B⁺-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 11.7 shows a typical node of a B⁺-tree. It contains up to $n - 1$ search-key values K_1, K_2, \dots, K_{n-1} , and n pointers P_1, P_2, \dots, P_n . The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$.

We consider first the structure of the **leaf nodes**. For $i = 1, 2, \dots, n - 1$, pointer P_i points to a file record with search-key value K_i . Pointer P_n has a special purpose that we shall discuss shortly.

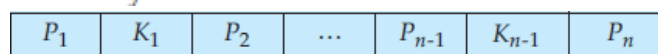


Figure 11.7 Typical node of a B⁺-tree.

Figure 11.8 shows one leaf node of a B⁺-tree for the *instructor* file, in which we have chosen n to be 4, and the search key is *name*.

Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $\lceil (n - 1)/2 \rceil$ values. With $n = 4$ in our example B⁺-tree, each leaf must contain at least 2 values, and at most 3 values.

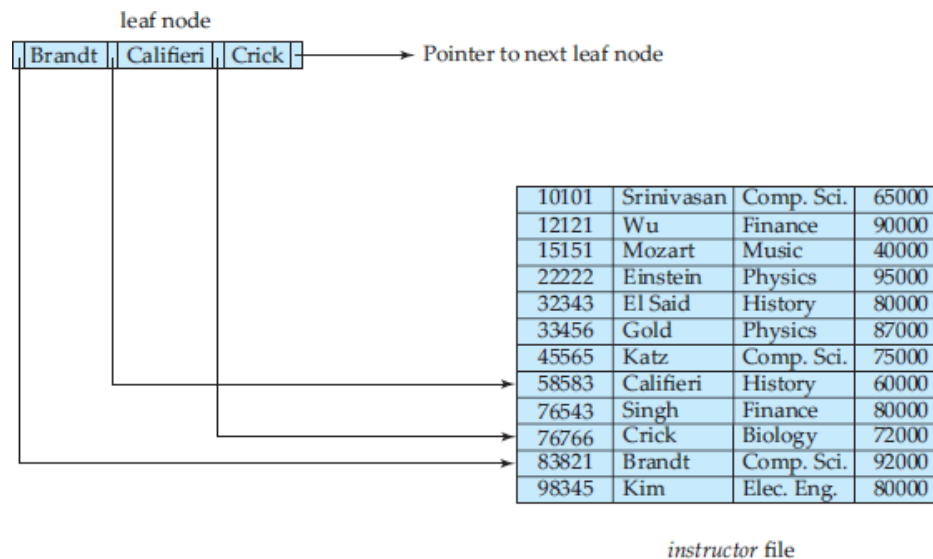


Figure 11.8 A leaf node for *instructor* B⁺-tree index ($n = 4$).

The **nonleaf nodes** of the B⁺-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and *must* hold at least $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fanout* of the node. Nonleaf nodes are also referred to as **internal nodes**.

Let us consider a node containing m pointers ($m \leq n$). For $i = 2, 3, \dots, m - 1$, pointer P_i points to the subtree that contains search-key values less than K_i and greater than or equal to K_{i-1} . Pointer P_m points to the part of the subtree that contains those key values greater than or equal to K_{m-1} , and pointer P_1 points to the part of the subtree that contains those search-key values less than K_1 .

Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B⁺-tree, for any n , that satisfies the preceding requirements.

Figure 11.9 shows a complete B⁺-tree for the *instructor* file (with $n = 4$). We have shown instructor names abbreviated to 3 characters in order to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.

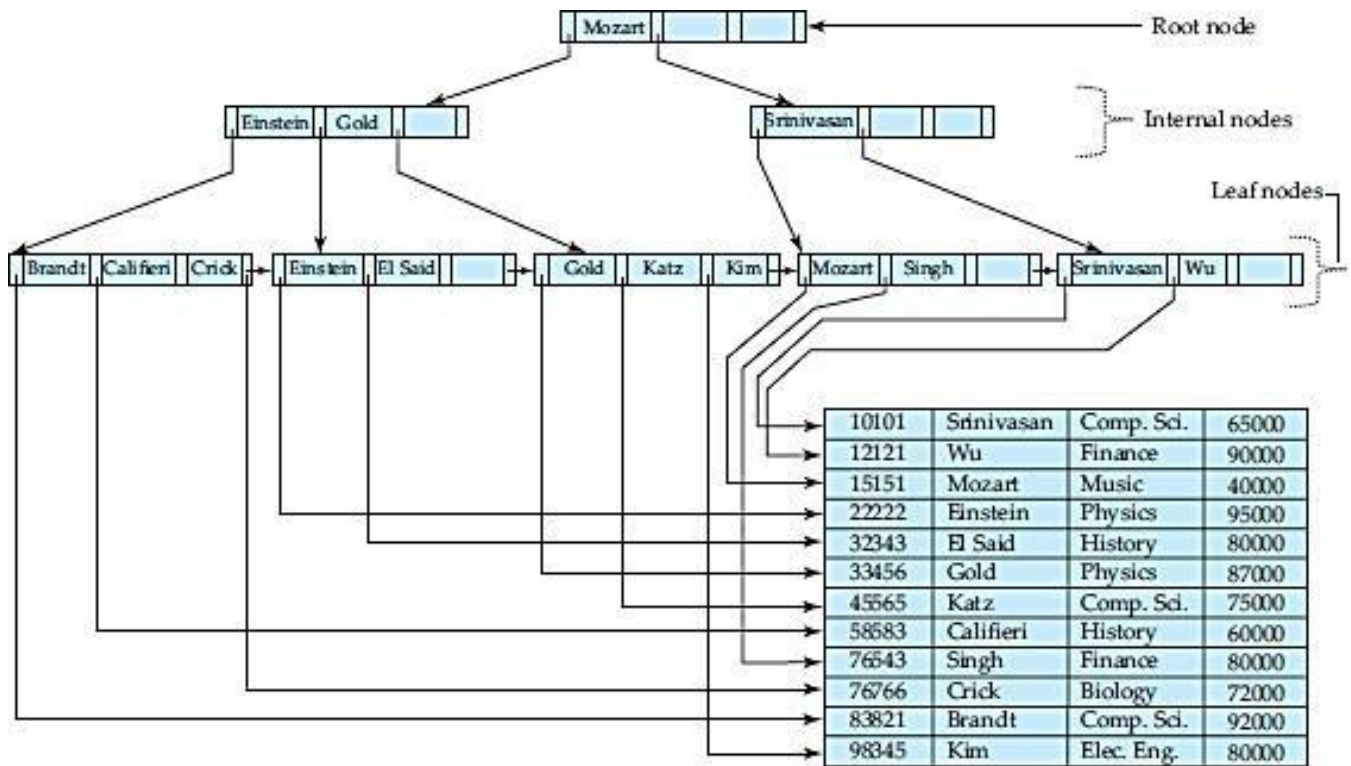


Figure 11.9 B⁺-tree for *instructor* file ($n = 4$).

Queries on B⁺ Trees

Let us consider how we process queries on a B⁺-tree. Suppose that we wish to find records with a search-key value of V .

Intuitively, the function starts at the root of the tree, and traverses the tree down until it reaches a leaf node that would contain the specified value if it exists in the tree. Specifically, starting with the root as the current node, the function repeats the following steps until a leaf node is reached. First, the current node is examined, looking for the smallest i such that search-key value K_i is greater than or equal to V . Suppose such a value is found; then, if K_i is equal to V , the current node is set to the node pointed to by P_{i+1} , otherwise $K_i > V$, and the current node is set to the node pointed to by P_i . If no such value K_i is found, then clearly $V > K_{m-1}$, where P_m is the last non null pointer in the node. In this case the current node is set to that pointed to by P_m . The above procedure is repeated, traversing down the tree until a leaf node is reached.

At the leaf node, if there is a search-key value equal to V , let K_i be the first such value; pointer P_i directs us to a record with search-key value K_i . The function then returns the leaf node L and the index i . If no search-key with value V is found in the leaf node, no record with key value V exists in the relation, and function find returns null, to indicate failure.


```

function find(value V)
/* Returns leaf node C and index i such that  $C.P_i$  points to first record
* with search key value V */
Set C = root node
while (C is not a leaf node) begin
    Let i = smallest number such that  $V \leq C.K_i$ 
    if there is no such number i then begin
        Let  $P_m$  = last non-null pointer in the node
        Set C =  $C.P_m$ 
    end
    else if ( $V = C.K_i$ )
        then Set C =  $C.P_{i+1}$ 
    else C =  $C.P_i$  /*  $V < C.K_i$  */
end
/* C is a leaf node */
Let i be the least value such that  $K_i = V$ 
if there is such a value i
    then return (C, i)
    else return null ; /* No record with key value V exists */

```

Updates on B⁺ Trees

When a record is inserted into, or deleted from a relation, indices on the relation must be updated correspondingly.

Insertion

We now consider an example of insertion in which a node must be split. Assume that a record is inserted on the *instructor* relation, with the *name* value being Adams. We then need to insert an entry for “Adams” into the B⁺-tree of Figure 11.9. Using the algorithm for lookup, we find that “Adams” should appear in the leaf node containing “Brandt”, “Califieri”, and “Crick.” There is no room in this leaf to insert the search-key value “Adams.” Therefore, the node is *split* into two nodes. Figure 11.12 shows the two leaf nodes that result from the split of the leaf node on inserting “Adams”. The search-key values “Adams” and “Brandt” are in one leaf, and “Califieri” and “Crick” are in the other. In general, we take the *n* search-key values (the *n* – 1 values in the leaf node plus the value being inserted), and put the first $\lceil n/2 \rceil$ in the existing node and the remaining values in a newly created node.

Having split a leaf node, we must insert the new leaf node into the B⁺-tree structure. In our example, the new node has “Califieri” as its smallest search-key value. We need to insert an entry with this search-key value, and a pointer to the new node, into the parent of the leaf node that was split. The B⁺-tree of Figure 11.13 shows the result of the insertion. It was possible to perform this insertion with no further node split, because there was room in the parent node for the new entry. If there were no room, the parent would have had to be split, requiring an entry to be added to its parent. In the worst case, all nodes along the path to the root must be split. If the root itself is split, the entire tree becomes deeper.

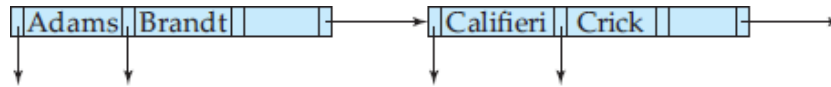


Figure 11.12 Split of leaf node on insertion of "Adams"

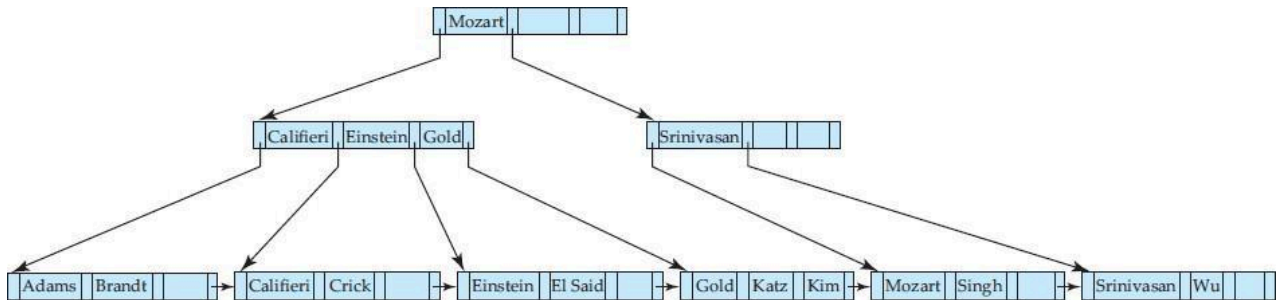


Figure 11.13 Insertion of "Adams" into the B⁺-tree of Figure 11.9.

```

procedure insert(value K, pointer P)
    if (tree is empty) create an empty leaf node L, which is also the root
    else Find the leaf node L that should contain key value K
    if (L has less than  $n - 1$  key values)
        then insert_in_leaf (L, K, P)
    else begin /* L has  $n - 1$  key values already, split it */
        Create node L'
        Copy  $L.P_1 \dots L.K_{n-1}$  to a block of memory T that can
            hold  $n$  (pointer, key-value) pairs
        insert_in_leaf (T, K, P)
        Set  $L'.P_n = L.P_n$ ; Set  $L.P_n = L'$ 
        Erase  $L.P_1$  through  $L.K_{n-1}$  from L
        Copy  $T.P_1$  through  $T.K_{\lceil n/2 \rceil}$  from T into L starting at  $L.P_1$ 
        Copy  $T.P_{\lceil n/2 \rceil + 1}$  through  $T.K_n$  from T into L' starting at  $L'.P_1$ 
        Let K' be the smallest key-value in L'
        insert_in_parent(L, K', L')
    end

procedure insert_in_leaf (node L, value K, pointer P)
    if ( $K < L.K_1$ )
        then insert P, K into L just before  $L.P_1$ 
    else begin
        Let  $K_i$  be the highest value in L that is less than K
        Insert P, K into L just after  $T.K_i$ 
    end
    
```

```

procedure insert_in_parent(node  $N$ , value  $K'$ , node  $N'$ )
  if ( $N$  is the root of the tree)
    then begin
      Create a new node  $R$  containing  $N$ ,  $K'$ ,  $N'$  /*  $N$  and  $N'$  are pointers */
      Make  $R$  the root of the tree
      return
    end
  Let  $P = \text{parent}(N)$ 
  if ( $P$  has less than  $n$  pointers)
    then insert ( $K'$ ,  $N'$ ) in  $P$  just after  $N$ 
  else begin /* Split  $P$  */
    Copy  $P$  to a block of memory  $T$  that can hold  $P$  and ( $K'$ ,  $N'$ )
    Insert ( $K'$ ,  $N'$ ) into  $T$  just after  $N$ 
    Erase all entries from  $P$ ; Create node  $P'$ 
    Copy  $T.P_1 \dots T.P_{\lceil n/2 \rceil}$  into  $P$ 
    Let  $K'' = T.K_{\lceil n/2 \rceil}$ 
    Copy  $T.P_{\lceil n/2 \rceil + 1} \dots T.P_{n+1}$  into  $P'$ 
    insert_in_parent( $P$ ,  $K''$ ,  $P'$ )
  end

```

Deletion

We now consider deletions that cause tree nodes to contain too few pointers. First, let us delete “Srinivasan” from the B^+ -tree of Figure 11.13. The resulting B^+ -tree appears in Figure 11.16. We now consider how the deletion is performed. We first locate the entry for “Srinivasan” by using our lookup algorithm. When we delete the entry for “Srinivasan” from its leaf node, the node is left with only one entry, “Wu”. Since, in our example, $n = 4$ and $1 < \lceil (n - 1)/2 \rceil$, we must either merge the node with a sibling node, or redistribute the entries between the nodes, to ensure that each node is at least half-full. In our example, the underfull node with the entry for “Wu” can be merged with its left sibling node. We merge the nodes by moving the entries from both the nodes into the left sibling, and deleting the now empty right sibling. Once the node is deleted, we must also delete the entry in the parent node that pointed to the just deleted node.

In our example, the entry to be deleted is (Srinivasan, $n3$), where $n3$ is a pointer to the leaf containing “Srinivasan”. (In this case the entry to be deleted in the nonleaf node happens to be the same value as that deleted from the leaf; that would not be the case for most deletions.) After deleting the above entry, the parent node, which had a search key value “Srinivasan” and two pointers, now has one pointer (the leftmost pointer in the node) and no search-key values. Since $1 < \lceil n/2 \rceil$ for $n = 4$, the parent node is underfull. (For larger n , a node that becomes underfull would still have some values as well as pointers.)

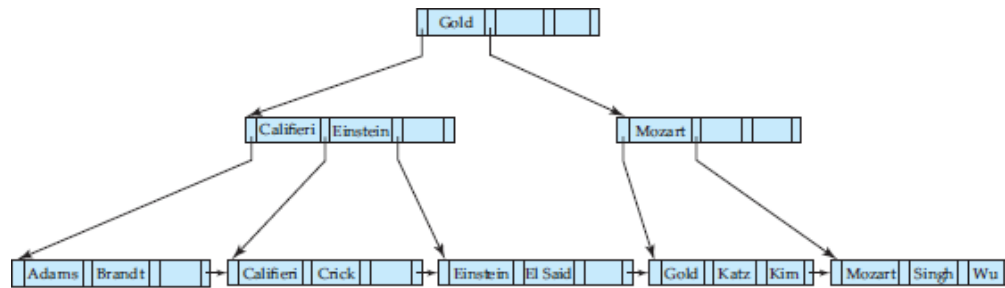


Figure 11.16 Deletion of “Srinivasan” from the B⁺-tree of Figure 11.13.

```

procedure delete(value K, pointer P)
    find the leaf node L that contains (K, P)
    delete_entry(L, K, P)

procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N' be the previous or next child of parent(N)
        Let K' be the value between pointers N and N' in parent(N)
        if (entries in N and N' can fit in a single node)
        then begin /* Coalesce nodes */
            if (N is a predecessor of N') then swap_variables(N, N')
            if (N is not a leaf)
            then append K' and all pointers and values in N to N'
            else append all (Ki, Pi) pairs in N to N'; set N'.Pn = N.Pn
            delete_entry(parent(N), K', N); delete node N
        end
    else begin /* Redistribution: borrow an entry from N' */
        if (N' is a predecessor of N) then begin
            if (N is a nonleaf node) then begin
                let m be such that N'.Pm is the last pointer in N'
                remove (N'.Km-1, N'.Pm) from N'
                insert (N'.Pm, K') as the first pointer and value in N,
                    by shifting other pointers and values right
                replace K' in parent(N) by N'.Km-1
            end
            else begin
                let m be such that (N'.Pm, N'.Km) is the last pointer/value
                    pair in N'
                remove (N'.Pm, N'.Km) from N'
                insert (N'.Pm, N'.Km) as the first pointer and value in N,
                    by shifting other pointers and values right
                replace K' in parent(N) by N'.Km
            end
        end
    else ... symmetric to the then case ...
    end
end
  
```


Hash Organization

For a huge database structure, it can be almost next to impossible to search all the index values through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate the direct location of a data record on the disk without using index structure.

Hashing uses hash functions with search keys as parameters to generate the address of a data record.

Hash Organization

- **Bucket** – A hash file stores data in bucket format. Bucket is considered a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function** – A hash function, **h**, is a mapping function that maps all the set of search-keys **K** to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided, the hash function always computes the same address. For example, if mod-4 hash function is used, then it shall generate only 5 values. The output address shall always be same for that function. The number of buckets provided remains unchanged at all times.

Operation

- **Insertion** – When a record is required to be entered using static hash, the hash function **h** computes the bucket address for search key **K**, where the record will be stored.

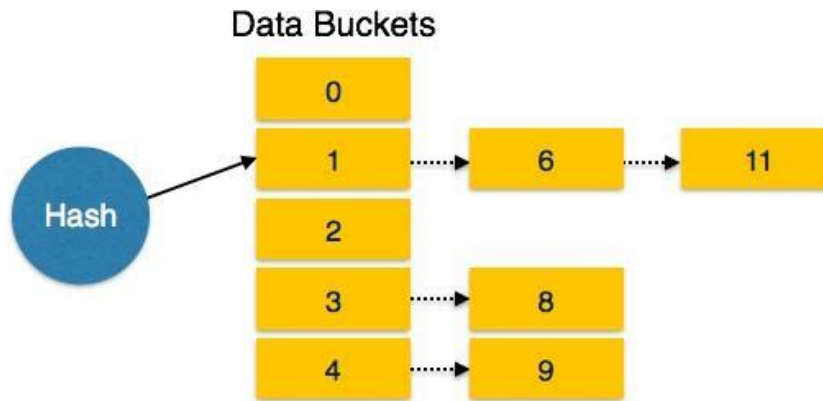
Bucket address = $h(K)$

- **Search** – When a record needs to be retrieved, the same hash function can be used to retrieve the address of the bucket where the data is stored.
- **Delete** – This is simply a search followed by a deletion operation.

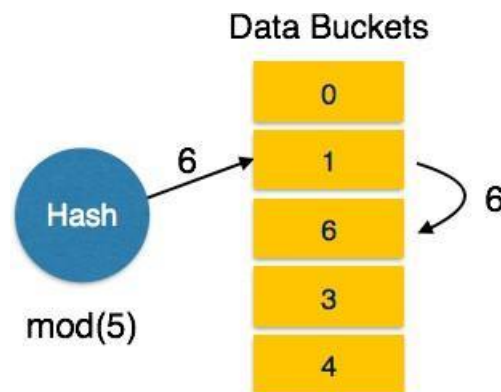
Bucket Overflow

The condition of bucket-overflow is known as **collision**. This is a fatal state for any static hash function. In this case, overflow chaining can be used.

- **Overflow Chaining** – When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called **Closed Hashing**.



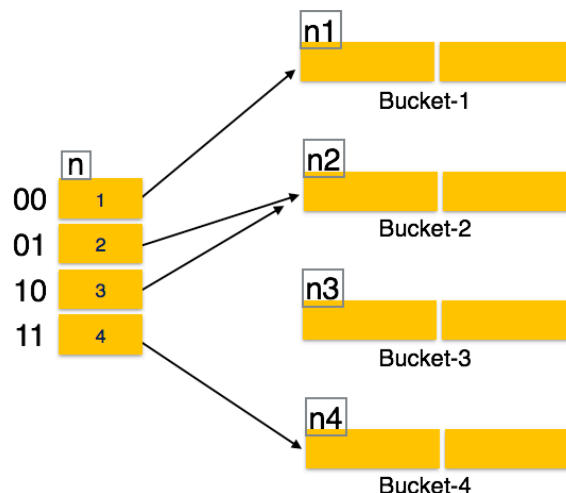
- **Linear Probing** – When a hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called **Open Hashing**.



Dynamic Hashing

The problem with static hashing is that it does not expand or shrink dynamically as the size of the database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as **extended hashing**.

Hash function, in dynamic hashing, is made to produce a large number of values and only a few are used initially.



COMPARISON OF THREE FILE ORGANIZATIONS

Refer text book

Database Management Systems, 3/e, Raghurama Krishnan, Johannes Gehrke, TMH

Chapter 8 File Organizations & Indexes

Pages from 232 to 236.

Review Questions

1. Explain about the measures that are to be considered for comparing the performance of various file organization techniques.
2. Explain in detail B+ tree file organization.
3. Write short notes on: i) Primary index ii) Clustered index iii) Secondary index.
4. Explain various anomalies that arise due to interleaved execution of transactions with suitable examples.
5. What is static hashing? What rules are followed for index selection?
6. Define transaction and explain desirable properties of transactions.
7. What is database Recovery? Explain Shadow paging in detail.
8. Explain about Conflict Serializability and view serializability.
9. Explain the following a) Concurrent executions, b) Transaction states.

References:

- Raghurama Krishnan, Johannes Gehrke, *Database Management Systems*, 3rd Edition, Tata McGraw Hill.
- C.J. Date, *Introduction to Database Systems*, Pearson Education.
- Elmasri Navrate, *Fundamentals of Database Systems*, Pearson Education.