K.D.K COLLEGE OF ENGINEEERING,NAGPUR

Database Management Systems

Unit 3

UNIT 3

3.1 OVERVIEW OF PHYSICAL STORAGE MEDIA

1. Several types of data storage exist in most computer systems. They vary in speed of access, cost per unit of data, and reliability.

   - **Cache:** most costly and fastest form of storage. Usually very small, and managed by the operating system.
   - **Main Memory (MM):** the storage area for data available to be operated on.
     - General-purpose machine instructions operate on main memory.
     - Contents of main memory are usually lost in a power failure or ``crash".
     - Usually too small (even with megabytes) and too expensive to store the entire database.
   - **Flash memory:** EEPROM (*electrically erasable programmable read-only memory*).
     - Data in flash memory survive from power failure.
     - Reading data from flash memory takes about 10 nano-secs (roughly as fast as from main memory), and writing data into flash memory is more complicated: write-once takes about 4-10 microsecs.
     - To overwrite what has been written, one has to first erase the entire bank of the memory. It may support only a limited number of erase cycles (10^4 to 10^6).
     - It has found its popularity as a replacement for disks for storing small volumes of data (5-10 megabytes).
   - **Magnetic-disk storage:** primary medium for long-term storage.
     - Typically the entire database is stored on disk.
     - Data must be moved from disk to main memory in order for the data to be operated on.
     - After operations are performed, data must be copied back to disk if any changes were made.
     - Disk storage is called **direct access** storage as it is possible to read data on the disk in any order (unlike sequential access).
     - Disk storage usually survives power failures and system crashes.
   - **Optical storage:** CD-ROM (compact-disk read-only memory), WORM (*write-once read-many*) disk (for archival storage of data), and *Juke box* (containing a few drives and numerous disks loaded on demand).
   - **Tape Storage:** used primarily for backup and archival data.
     - Cheaper, but much slower access, since tape must be read sequentially from the beginning.
     - Used as protection from disk failures!

2. The storage device hierarchy is presented in Figure 3.1, where the higher levels are expensive (cost per bit), fast (access time), but the capacity is smaller.

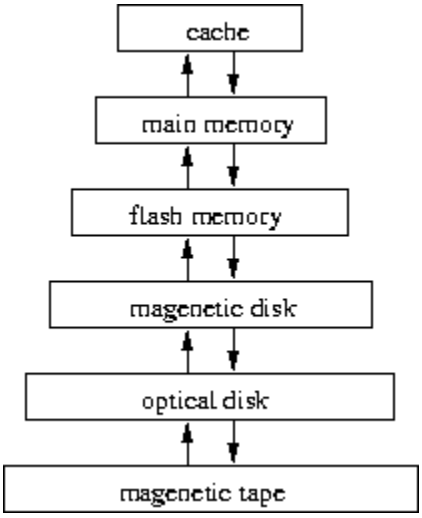


Figure 3.1: Storage-device hierarchy

3. Another classification: Primary, secondary, and tertiary storage.
    1. Primary storage: the fastest storage media, such as cash and main memory.
    2. Secondary (or on-line) storage: the next level of the hierarchy, e.g., magnetic disks.
    3. Tertiary (or off-line) storage: magnetic tapes and optical disk juke boxes.

Volatility of storage. *Volatile storage* loses its contents when the power is removed. Without power backup, data in the volatile storage (the part of the hierarchy from main memory up) must be written to nonvolatile storage for safekeeping.

3.2 INDEXING and HASHING

1. Many queries reference only a small proportion of records in a file. For example, finding all records at Perryridge branch only returns records where *bname* = ``Perryridge".
2. We should be able to locate these records directly, rather than having to read **every** record and check its

Basic Concepts

1. An index for a file works like a catalogue in a library. Cards in alphabetic order tell us where to find books by a particular author.
2. In real-world databases, indices like this might be too large to be efficient. We'll look at more sophisticated indexing techniques.
3. There are two kinds of indices.
    - Ordered indices: indices are based on a sorted ordering of the values.

- Hash indices: indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function, called a *hash function.*

We will consider several indexing techniques. No one technique is the best. Each technique is best suited for a particular database application.

Methods will be evaluated on:

Access Types -- types of access that are supported efficiently, e.g., value-based search or range search.

Access Time -- time to find a particular data item or set of items.

Insertion Time -- time taken to insert a new data item (includes time to find the right place to insert).

Deletion Time -- time to delete an item (includes time taken to find item, as well as to update the index structure).

Space Overhead -- additional space occupied by an index structure.

We may have more than one index or hash function for a file. (The library may have card catalogues by author, subject or title.)

The attribute or set of attributes used to look up records in a file is called the **search key** (not to be confused with primary key, etc.).

3.2.1 Ordered Indices

1. In order to allow fast **random** access, an index structure may be used.
2. A file may have several indices on different search keys.
3. If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the **primary index**, or **clustering index**. Note: The search key of a primary index is usually the primary key, but it is not necessarily so.
4. Indices whose search key specifies an order different from the sequential order of the file are called the **secondary indices**, or **nonclustering indices**.

- Primary Index
  - Dense and Sparse Indices
  - Multi-Level Indices
  - Index Update
- Secondary Indices

3.2.2 Primary Index

1. *Index-sequential files*: Files are ordered sequentially on some search key, and a primary index is associated with it.

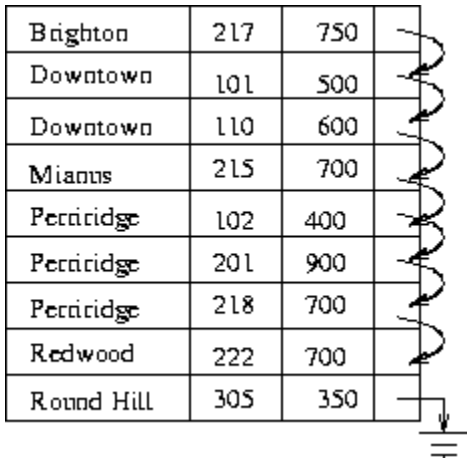


Figure 4.1: Sequential file for *deposit* records.

All files are ordered sequentially on some search key. Such files, with a primary index on the search key, are called **index-sequential files**.

An **index record**, or **index entry**, 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 and Sparse Indices

1. There are Two types of ordered indices:

   Dense Index:

   - An index record appears for **every** search key value in file.
   - This record contains search key value and a pointer to the actual record.

   Sparse Index:

   - Index records are created only for **some** of the records.
   - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.

- We start at that record pointed to by the index record, and proceed along the pointers in the file (that is, sequentially) until we find the desired record.

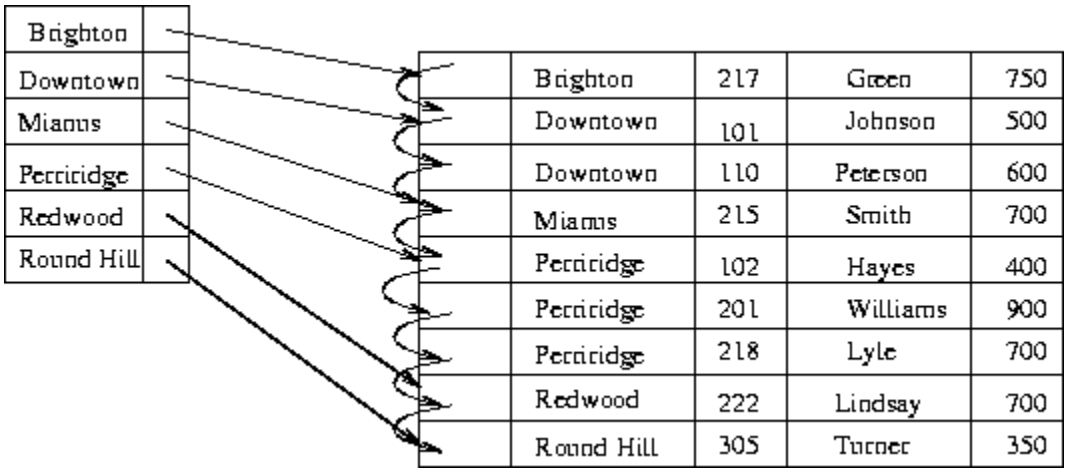


Figure: Dense index.

2. Notice how we would find records for Perryridge branch using both methods. (Do it!)

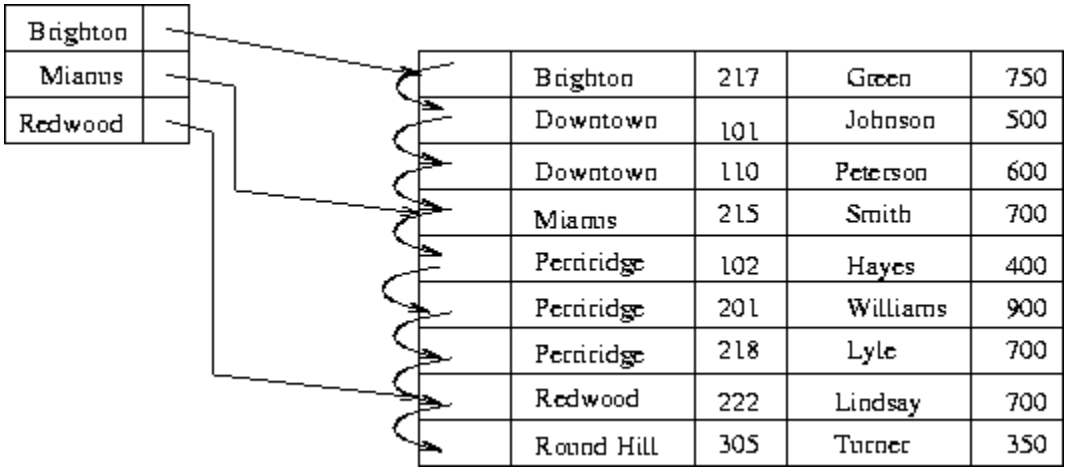


Figure: Sparse index.

3. Dense indices are faster in general, but sparse indices require less space and impose less maintenance for insertions and deletions. (Why?)
4. A good compromise: to have a sparse index with one entry per block.

   Why is this good?

   - Biggest cost is in bringing a block into main memory.
   - We are guaranteed to have the correct block with this method, unless record is on an overflow block (actually could be **several** blocks).
   - Index size still small.

3.2.3Multi-Level Indices

1. Even with a sparse index, index size may still grow too large. For 100,000 records, 10 per block, at one index record per block, that's 10,000 index records! Even if we can fit 100 index records per block, this is 100 blocks.
2. If index is too large to be kept in main memory, a search results in several disk reads.
   - If there are no overflow blocks in the index, we can use binary search.
   - This will read as many as $1 + \log_2(b)$ blocks (as many as 7 for our 100 blocks).
   - If index has overflow blocks, then sequential search typically used, reading all *b* index blocks.
3. Solution: Construct a sparse index on the index

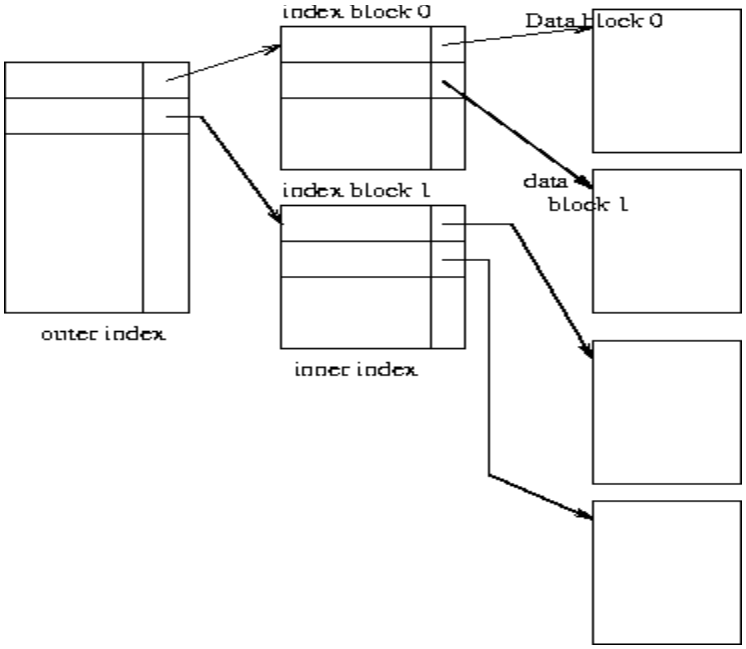


Figure: Two-level sparse index.

4. Use binary search on outer index. Scan index block found until correct index record found. Use index record as before - scan block pointed to for desired record.
5. For very large files, additional levels of indexing may be required.
6. Indices must be updated at all levels when insertions or deletions require it.
7. Frequently, each level of index corresponds to a unit of physical storage (e.g. indices at the level of track, cylinder and disk).

3.2.4 Secondary Indices

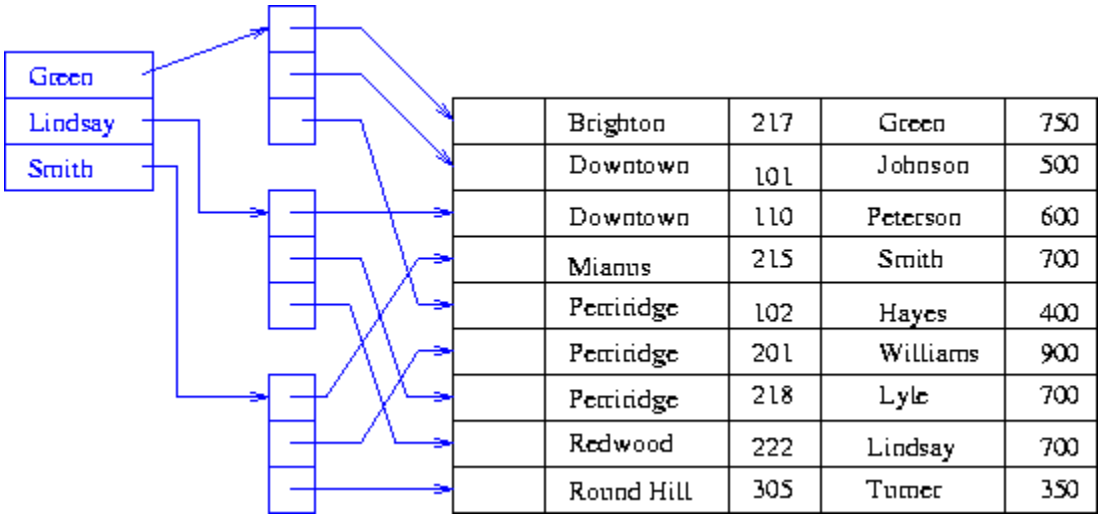
1. If the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value because the remaining records with the same search-key value could be anywhere in the file. Therefore, a secondary index must contain pointers to all the records.
2. 

Figure:Sparse secondary index on cname.

3. We can use an extra-level of indirection to implement secondary indices on search keys that are not candidate keys. A pointer does not point directly to the file but to a bucket that contains pointers to the file.
    - See Figure on secondary key *cname*.
    - To perform a lookup on Peterson, we must read all three records pointed to by entries in bucket 2.
    - Only one entry points to a Peterson record, but three records need to be read.
    - As file is not ordered physically by *cname*, this may take 3 block accesses.
4. Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file.
5. Secondary indices improve the performance of queries on non-primary keys.
6. They also impose serious overhead on database modification: whenever a file is updated, every index must be updated.
7. Designer must decide whether to use secondary indices or not.

3.2.5 B+-Tree Index Files

1. Primary disadvantage of index-sequential file organization is that performance degrades as the file grows. This can be remedied by costly re-organizations.
2. B + -tree file structure maintains its efficiency despite frequent insertions and deletions. It imposes some acceptable update and space overheads.
3. A B + -tree index is a *balanced tree* in which every path from the root to a leaf is of the same length.
4. Each nonleaf node in the tree must have between $\lceil n/2 \rceil$ and *n* children, where *n* is fixed for a particular tree.

Structure of a B+-Tree

1. A B + -tree index is a *multilevel* index but is structured differently from that of multi-level index sequential files.
2. A typical node (contains up to *n*-1 search key values $K_1, K_2, \ldots, K_{n-1}$, and *n* pointers $P_1, P_2, \ldots, P_n$. Search key values in a node are kept in sorted order.

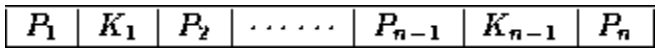


Figure 4.6 Typical node of a B+-tree.

3. For leaf nodes, P_i ($i = 1, \ldots, n-1$) points to either a file record with search key value K_i, or a bucket of pointers to records with that search key value. Bucket structure is used if search key is not a primary key, and file is not sorted in search key order.

   Pointer P_n (*n*th pointer in the leaf node) is used to chain leaf nodes together in linear order (search key order). This allows efficient **sequential** processing of the file.

   The range of values in each **leaf** do not overlap.

4. Non-leaf nodes form a multilevel index on leaf nodes.

   A non-leaf node may hold up to *n* pointers and must hold $\lceil n/2 \rceil$ pointers. The number of pointers in a node is called the *fan-out* of the node.

   Consider a node containing *m* pointers. Pointer P_i ($i = 2, \ldots, m$) points to a subtree containing search key values $\geq K_{i-1}$ and $< K_i$. Pointer P_m points to a subtree containing search key values $\geq K_{m-1}$. Pointer P_1 points to a subtree containing search key values $< K_1$.

5. Fig show B + -trees for the *deposit* file with *n*=3 and *n*=5.

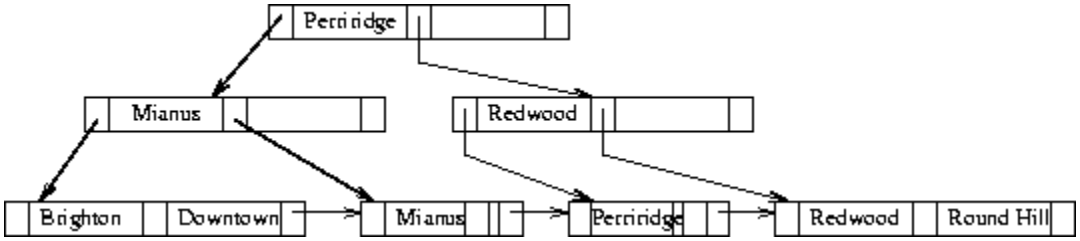


Figure:B+-tree for deposit file with n = 3

3.2.6: Static Hashing

1. Index schemes force us to traverse an index structure. Hashing avoids this.

- Hash File Organization
  - Hash Functions
  - Handling of bucket overflows
- Hash Indices

3.2.7 Hash File Organization

1. **Hashing** involves computing the address of a data item by computing a function on the search key value.
2. A **hash function h** is a function from the set of all search key values *K* to the set of all bucket addresses *B*.
   - We choose a number of buckets to correspond to the number of search key values we will have stored in the database.
   - To perform a lookup on a search key value K_i, we compute $h(K_i)$, and search the bucket with that address.
   - If two search keys *i* and *j* map to the same address, because $h(K_i) = h(K_j)$, then the bucket at the address obtained will contain records with both search key values.
   - In this case we will have to check the search key value of every record in the bucket to get the ones we want.
   - Insertion and deletion are simple

Hash Functions

1. A good hash function gives an average-case lookup that is a small constant, independent of the number of search keys.

2. We hope records are distributed uniformly among the buckets.
3. The worst hash function maps all keys to the same bucket.
4. The best hash function maps all keys to distinct addresses.
5. Ideally, distribution of keys to addresses is uniform and random.
6. Suppose we have 26 buckets, and map names beginning with *i*th letter of the alphabet to the *i*th bucket.
    - Problem: this does not give uniform distribution.
    - Many more names will be mapped to ``A" than to ``X".
    - Typical hash functions perform some operation on the internal binary machine representations of characters in a key.
    - For example, compute the sum, modulo # of buckets, of the binary representations of characters of the search key.
    - See Figure 11.18, using this method for 10 buckets (assuming the *i*th character in the alphabet is represented by integer *i*).

3.2.8 Handling of Bucket Overflows

1. **Open** hashing occurs where records are stored in different buckets. Compute the hash function and search the corresponding bucket to find a record.
2. **Closed** hashing occurs where all records are stored in **one** bucket. Hash function computes addresses within that bucket. (Deletions are difficult.) Not used much in database applications.
3. **Drawback to our approach:** Hash function must be chosen at implementation time.
    - Number of buckets is fixed, but the database may grow.
    - If number is too large, we waste space.
    - If number is too small, we get too many ``collisions", resulting in records of many search key values being in the same bucket.
    - Choosing the number to be twice the number of search key values in the file gives a good space/performance tradeoff.

Hash Indices

 A hash index organizes the search keys with their associated pointers into a hash file structure.

 We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).

 Strictly speaking, hash indices are only secondary index structures, since if a file itself is organized using hashing, there is no need for a separate hash index structure on it.

Dynamic Hashing

1. As the database grows over time, we have three options:
    - Choose hash function based on current file size. Get performance degradation as file grows.
    - Choose hash function based on anticipated file size. Space is wasted initially.
    - Periodically re-organize hash structure as file grows. Requires selecting new hash function, recomputing all addresses and generating new bucket assignments. Costly, and shuts down database.
2. Some hashing techniques allow the hash function to be modified dynamically to accommodate the growth or shrinking of the database. These are called **dynamic hash functions**.
    - **Extendable hashing** is one form of dynamic hashing.
    - Extendable hashing splits and coalesces buckets as database size changes.
    - This imposes some performance overhead, but space efficiency is maintained.
    - As reorganization is on one bucket at a time, overhead is acceptably low.

Handling of Bucket Overflows

So far, we have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur. Bucket overflow can occur for several reasons:

• **Insufficient buckets**. The number of buckets, which we denote *nB*, must be chosen such that $nB > nr/fr$, where *nr* denotes the total number of records that will be stored, and *fr* denotes the number of records that will fit in a bucket. This designation, of course, assumes that the total number of records is known when the hash function is chosen.

• **Skew**. Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:

1. Multiple records may have the same search key.

2. The chosen hash function may result in nonuniform distribution of search keys.

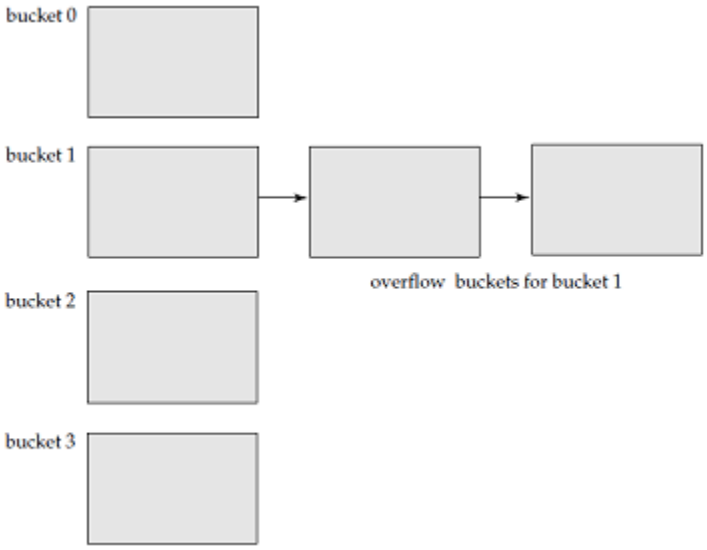
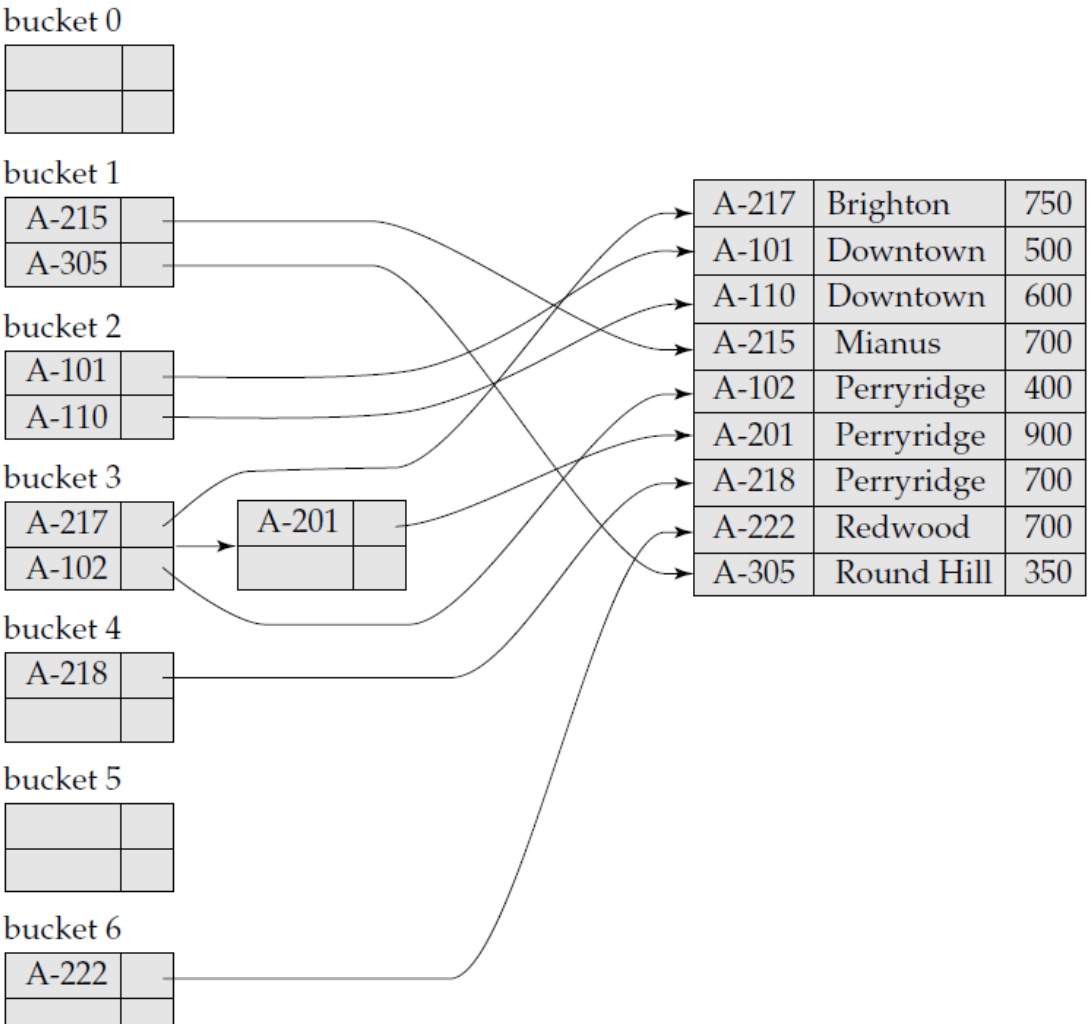


Fig:Bucket Overflow

The system must examine all the records in bucket *b* to see whether they match the search key, as before. In addition, if bucket *b* has overflow buckets, the system must examine the records in all the overflow buckets also. The form of hash structure that we have just described is sometimes referred to as **closed hashing**. Under an alternative approach, called **open hashing**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets *B*. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*.
Other policies, such as computing further hash functions, are also used. Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems. The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only
lookup and insertion operations on their symbol tables. However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

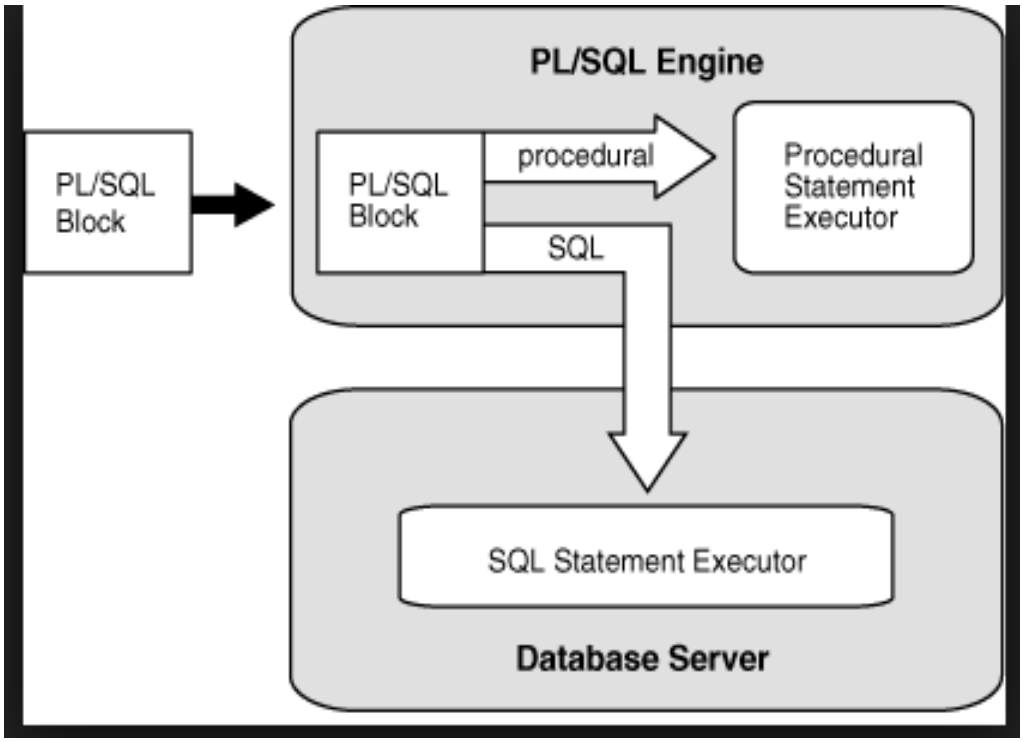
3.2.9 Hash Indices:

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 pointers, into a
hash file structure. We construct a hash index as follows. We apply a hash function
on a search key to identify a bucket, and store the key and its associated pointers
in the bucket (or in overflow buckets).



We use the term *hash index* to denote hash file structures as well as secondary hash indices. Strictly speaking, hash indices are only secondary index structures. A hash index is never needed as a primary index structure, since, if a file itself is organized by hashing, there is no need for a separate hash index structure on it. However,
since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a primary hash index on it.

PL/SQL is *procedural* language extension to SQL



Architecture of PL/SQL

Advantages:

- Tight Integration with SQL
- High Performance
- Full portability
- Tight security
- Support for developing web applications

A PL/SQL unit is any one of the following:

- PL/SQL block
- Function
- Procedure
- Trigger

PL/SQL Block consists of three sections

- The Declaration section (optional).

- The Execution section (mandatory).
- The Exception (or Error) section (optional).

PL/SQL block has the following structure:

DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCEPTION
    Exception-handling statements
END ;

- The *declaration section* is the first section of the PL/SQL block.
- It contains definitions of PL/SQL identifiers such as variables, constants and so on.

Example

```
DECLARE
    v_first_name VARCHAR(35) ;
    v_last_name VARCHAR(35) ;
    v_counter NUMBER := 0 ;
```

- The executable section is the next section of the PL/SQL block.
- This section contains executable statements that allow you to manipulate the variables that have been declared in the declaration section.

```
BEGIN
    SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = 123 ;
    DBMS_OUTPUT.PUT_LINE
    ('Student name :' || v_first_name ||'  '|| v_last_name);
END;
```

- The *exception-handling section* is the last section of the PL/SQL block.
- This section contains statements that are executed when a runtime error occurs within a block.

i) NO_DATA_FOUND
ii) LOGIN_DENIED
iii) ZERO_DIVIDE

```
Eg: EXCEPTION
    WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE
(' There is no student with student id 123 ');
END;
```

Example:

```
DECLARE
        v_first_name VARCHAR(35);
        v_last_name VARCHAR(35);
BEGIN
        SELECT first_name, last_name
        INTO v_first_name, v_last_name
        FROM student
        WHERE student_id = 123;
        DBMS_OUTPUT.PUT_LINE
        ('Student name: '||v_first_name||' '||v_last_name);
EXCEPTION
                WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE
                ('There is no student with student id 123');
END;
```

A stored Procedure :

A stored procedure is a named PL/SQL block which performs one or more specific task.

Procedures: Passing Parameters

We can pass parameters to procedures in three ways.

1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

A procedure may or may not return any value.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter_name [IN | OUT | IN
    OUT] type [, ...])]

 {IS | AS}
```

BEGIN

< procedure_body >

END;

Example for IN parameter:

```
CREATE PROCEDURE Country_procedure(IN countryName VARCHAR(255))
 BEGIN
      SELECT *
      FROM country
      WHERE name = countryName;
END
```

Execute: call country_procedure('India')

Example Out Parameter:

```
CREATE PROCEDURE Cpopulation(
 IN cname VARCHAR(25),
 OUT tpopulation INT)
BEGIN
        SELECT population
         INTO tpopulation
         FROM country
         WHERE name = cname;
END
```

Functions:

A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

```
CREATE [OR REPLACE] FUNCTION function_name [parameters] RETURN return_datatype;
IS
Declaration_section
BEGIN
Execution_section
Return return_variable;
EXCEPTION
exception section
Return return_variable;
END;
```

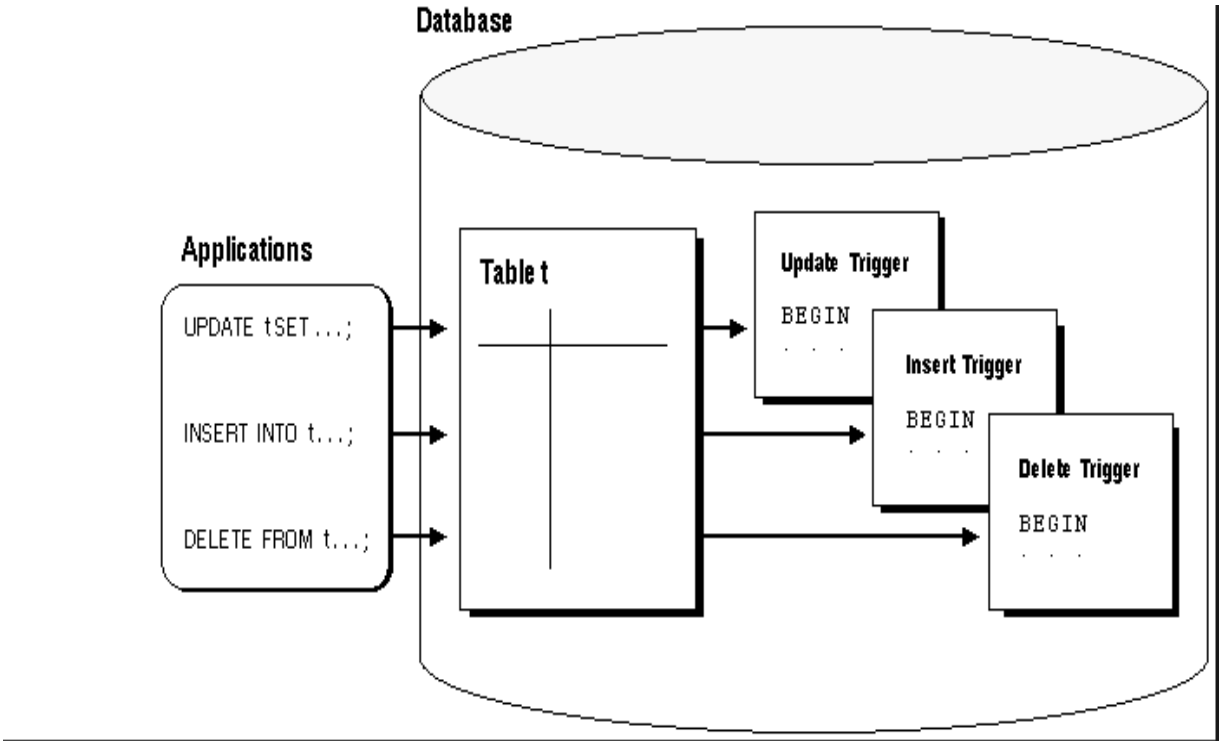
Example:

```
CREATE FUNCTION PopulationLevel(population int) RETURNS VARCHAR(10)
  BEGIN
    DECLARE lvl varchar(10);
     IF population > 50000 THEN
 SET lvl = 'HIGH';
    ELSEIF (population <= 50000 AND population >= 10000) THEN
       SET lvl = 'MEDIUM';
    ELSEIF population < 10000 THEN
       SET lvl = 'LOW';
    END IF;
  RETURN (lvl);
END
```

Calling Function: select name, populationLevel(population) from country;

Triggers:

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed



Syntax for Creating a Trigger:

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF } {INSERT [OR] | UPDATE [OR] | DELETE}
ON table_name
[FOR EACH ROW]
WHEN (condition)
BEGIN
--- sql statements
END;
```

Example:

```
CREATE TRIGGER before_city_update
   BEFORE UPDATE ON city
   FOR EACH ROW
BEGIN
   INSERT INTO city_audit
```

```
    SET action = 'update',
     id = OLD.id,
       name = old.name,
        changedat = NOW();
END
```

```
1) CREATE TABLE city_audit (
    id1 INT AUTO_INCREMENT PRIMARY KEY,
    id INT NOT NULL,
    name VARCHAR(50) NOT NULL,
    changedat DATETIME DEFAULT NULL,
    action VARCHAR(50) DEFAULT NULL
);
2) update city set population='1000' where name='Gaza';
3) mysql> select * from city_audit;
+-----+------+------+---------------------+--------+
| id1 | id   | name | changedat           | action |
+-----+------+------+---------------------+--------+
|   1 | 4074 | Gaza | 2016-08-30 05:44:26 | update |
+-----+------+------+---------------------+--------+
1 row in set (0.00 sec)
```

Example:

```
Select * from customers;

+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
   sal_diff number;
BEGIN
   sal_diff := :NEW.salary  - :OLD.salary;
   dbms_output.put_line('Old salary: ' || :OLD.salary);
   dbms_output.put_line('New salary: ' || :NEW.salary);
   dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

```
Old salary:
New salary: 7500
Salary difference:
```

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

An assertion is a statement in SQL that ensures a certain condition will always exist in the database. Assertions are like column and table constraints, except that they are specified separately from table definitions.

- An assertion is a predicate expressing a condition we wish the database to always satisfy.
- Domain constraints, functional dependency and referential integrity are special forms of assertion.
- Where a constraint cannot be expressed in these forms, we use an assertion, e.g.

  - Ensuring the sum of loan amounts for each branch is less than the sum of all account balances at the branch.
  - Ensuring every loan customer keeps a minimum of $1000 in an account.

- An assertion in DQL-92 takes the form,

create assertion assertion-name check predicate