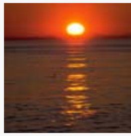
Chapter 3: SQL

Database System Concepts, 5th Ed.



Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

Data Definition Language

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.
- More are covered in Chapter 4.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i

- Example:

  create table *branch*
  (*branch_name* char(15) **not null,**
  branch_city char(30),
  assets integer)

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)

Example: Declare *branch_name* as the primary key for *branch*
.

create table *branch*
(*branch_name* char(15)**,**
branch_city char(30),
assets integer,
primary key (*branch_name*))

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89

Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation:

  alter table *r* **add** *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned *null* as the value for the new attribute.
- The **alter table** command can also be used to drop attributes of a relation:

  alter table *r* **drop** *A*

  where *A* is the name of an attribute of relation *r*
  - Dropping of attributes not supported by many databases

Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n} (\sigma_P (r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all branches in the *loan* relation:

  select *branch_name*
  from *loan*

- In the relational algebra, the query would be:

$$\Pi_{branch\_name}(loan)$$

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the names of all branches in the *loan* relations, and remove duplicates

  select distinct *branch_name*
  from *loan*

- The keyword **all** specifies that duplicates not be removed.

  select all *branch_name*
  from *loan*

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *loan*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
- The query:

  select *loan_number, branch_name, amount* ∗ 100
  from *loan*

  would return a relation that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100.

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

  select *loan_number*
  from *loan*
  where *branch_name* = 'Perryridge' **and** *amount* > 1200

- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

The where Clause (Cont.)

- SQL includes a **between** comparison operator
- Example: Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)

select *loan_number*
from *loan*
where *amount* **between** 90000 **and** 100000

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower X loan*

  select *
  from *borrower, loan*

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

  select *customer_name, borrower.loan_number, amount*
  from *borrower, loan*
  where *borrower.loan_number = loan.loan_number* **and**
  branch_name = 'Perryridge'

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id.*

select *customer_name, borrower.loan_number* **as** *loan_id, amount*
from *borrower, loan*
where *borrower.loan_number* = *loan.loan_number*

Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

    select *customer_name, T.loan_number, S.amount*
    from *borrower* **as** *T, loan* **as** *S*
    where *T.loan_number = S.loan_number*

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

    select distinct *T.branch_name*
    from *branch* **as** *T, branch* **as** *S*
    where *T.assets > S.assets* **and** *S.branch_city* = 'Brooklyn'

- Keyword **as** is optional and may be omitted

    borrower **as** *T* ≡ *borrower T*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring “Main”.

  select *customer_name*
from *customer*
where *customer_street* **like** '% Main%'

- Match the name “Main%”

  like 'Main\%' **escape** '\'

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan in Perryridge branch

  select distinct *customer_name*
  from *borrower, loan*
  where *borrower loan_number* = *loan.loan_number* **and**
  branch_name = 'Perryridge'
  order by *customer_name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *customer_name* **desc**

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. $\boldsymbol{\sigma_\theta (r_1)}$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta (r_1)$.
2. $\boldsymbol{\Pi_A (r)}$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A (t_1)$ in $\Pi_A (r_1)$ where $\Pi_A (t_1)$ denotes the projection of the single tuple t_1.
3. $\boldsymbol{r_1 \times r_2}$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1 . t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

- Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  $r_1 = \{(1, a)\ (2,a)\}$ $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1) \times r_2$ would be

  $\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$

- SQL duplicate semantics:

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where P

  is equivalent to the *multiset* version of the expression:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations $\cup, \cap, -$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all**, **intersect all** and **except all.**

  Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:

  - $m + n$ times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, $m - n$) times in *r* **except all** *s*

Set Operations

- Find all customers who have a loan, an account, or both:

  (select *customer_name* **from** *depositor*)
  union
  (select *customer_name* **from** *borrower)*

- Find all customers who have both a loan and an account.

  (**select** *customer_name* **from** *depositor*)
  intersect
  (select *customer_name* **from** *borrower)*

- Find all customers who have an account but no loan.

  (**select** *customer_name* **from** *depositor*)
  except
  (select *customer_name* **from** *borrower)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

  select avg *(balance)*
  from *account*
  where *branch_name* = 'Perryridge'

- Find the number of tuples in the *customer* relation.

  select count (*)
  from *customer*

- Find the number of depositors in the bank.

  select count (distinct *customer_name)*
  from *depositor*

Aggregate Functions – Group By

- Find the number of depositors for each branch.

select *branch_name,* **count (distinct** *customer_name)*
from *depositor, account*
where *depositor.account_number = account.account_number*
group by *branch_name*

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than $1,200.

> **select** *branch_name,* **avg** (*balance*)
> **from** *account*
> **group by** *branch_name*
> **having avg** (*balance*) > 1200

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.
  - Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

    select *loan_number*
    from *loan*
    where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls
  - More on next slide

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null   or   null <> null   or   null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*)   = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown)* = *unknown*
  - AND: *(true* **and** *unknown)* = *unknown,*
    (false **and** *unknown)* = *false,*
    (unknown **and** *unknown)* = *unknown*
  - NOT*: (***not** *unknown)* = *unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)
  from *loan*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find all customers who have both an account and a loan at the bank.

select distinct *customer_name*
from *borrower*
where *customer_name* **in** (**select** *customer_name*
from *depositor*)

- Find all customers who have a loan at the bank but do not have an account at the bank

select distinct *customer_name*
from *borrower*
where *customer_name* **not in** (**select** *customer_name*
from *depositor*)

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
         branch_name = 'Perryridge'  and
         (branch_name, customer_name ) in
                (select branch_name, customer_name
                 from depositor, account
                 where depositor.account_number =
                     account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

- Find all branches that have greater assets than some branch located in Brooklyn.

select distinct *T.branch_name*
from *branch* **as** *T, branch* **as** *S*
where *T.assets* > *S.assets* **and**
S.branch_city = 'Brooklyn'

- Same query using > **some** clause

select *branch_name*
from *branch*
where *assets* > **some**
(select *assets*
from *branch*
where *branch_city* = 'Brooklyn')

Definition of Some Clause

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}$) = true (read: 5 < some tuple in the relation)

(5 < **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = false

(5 = **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true

(5 $\neq$ **some** $\begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}$) = true (since 0 $\neq$ 5)

(= **some**) $\equiv$ **in**

However, ($\neq$ **some**) $\not\equiv$ **not in**

Example Query

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

select *branch_name*
    from *branch*
    where *assets* > **all**
        (select *assets*
        from *branch*
        where *branch_city* = 'Brooklyn')

Definition of all Clause

- F <comp> **all** *r* ⇔ ∀ *t* ∈ *r* (F <comp> *t*)

$$\left(5 < \textbf{all} \ \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}\right) = \text{false}$$

$$\left(5 < \textbf{all} \ \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline \end{array}\right) = \text{true}$$

$$\left(5 = \textbf{all} \ \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline \end{array}\right) = \text{false}$$

$$\left(5 \neq \textbf{all} \ \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline \end{array}\right) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$$

(≠ **all**) ≡ **not in**

However, (= **all**) ≢ **in**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Example Query

- Find all customers who have an account at all branches located in Brooklyn.

```
select distinct S.customer_name
          from depositor as S
          where not exists (
                    (select branch_name
                    from branch
                    where branch_city = 'Brooklyn')
                    except
                    (select R.branch_name
                    from depositor as T, account as R
                    where T.account_number = R.account_number and
                              S.customer_name = T.customer_name ))
```

- Note that $X - Y = \emptyset \iff X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

select *T.customer_name*
from *depositor* **as** *T*
where unique (
  select *R.customer_name*
  from *account, depositor* **as** *R*
  where *T.customer_name* = *R.customer_name* **and**
    R.account_number = *account.account_number* **and**
    account.branch_name = 'Perryridge')

Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

```
select distinct T.customer_name
from depositor  as T
where not unique (
      select R.customer_name
      from account, depositor as R
      where T.customer_name = R.customer_name and
            R.account_number = account.account_number  and
            account.branch_name = 'Perryridge')
```

- Variable from outer level is known as a **correlation variable**

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average account balance of those branches where the average account balance is greater than $1200.

  select *branch_name, avg_balance*
  from (**select** *branch_name,* **avg** (*balance*)
          from *account*
          group by *branch_name*)
          as *branch_avg* (*branch_name, avg_balance*)
  where *avg_balance* > 1200

  Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all accounts with the maximum balance

```
with max_balance (value) as
    select max (balance)
    from account
select account_number
from account, max_balance
where account.balance = max_balance.value
```

Complex Queries using With Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
with branch_total (branch_name, value) as
        select branch_name, sum (balance)
        from account
        group by branch_name
with branch_total_avg (value) as
        select avg (value)
        from branch_total
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_total_avg.value
```

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

  (**select** *customer_name, borrower.loan_number, branch_name*
  from *borrower, loan*
  where *borrower.loan_number* = *loan.loan_number*)

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.

Example Queries

- A view consisting of branches and their customers

```
create view all_customer as
        (select branch_name, customer_name
         from depositor, account
        where depositor.account_number =
                    account.account_number )
        union
        (select branch_name, customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number )
```

- Find all customers of the Perryridge branch

```
    select customer_name
            from all_customer
            where branch_name = 'Perryridge'
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
      Find any view relation v_i in e_1
      Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

  delete from *account*
  where *branch_name* = 'Perryridge'

- Delete all accounts at every branch located in the city ‘Needham’.

  delete from *account*
  where *branch_name* **in** (**select** *branch_name*
  from *branch*
  where *branch_city* = 'Needham')

Example Query

- Delete the record of all accounts with balances below the average at the bank.

delete from *account*
    where *balance* < (**select avg** (*balance*)
        from *account*)

- Problem: as we delete tuples from deposit, the average balance changes
- Solution used in SQL:
  1. First, compute **avg** balance and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *account*

  insert into *account*
  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)
  values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account*
  values ('A-777','Perryridge', *null*)

Modification of the Database – Insertion

- Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

  insert into *account*
      select *loan_number, branch_name,* 200
      from *loan*
      where *branch_name* = 'Perryridge'
  insert into *depositor*
      select *customer_name, loan_number*
      from *loan, borrower*
      where branch_name = 'Perryridge'
          and *loan.account_number* = *borrower.account_number*

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like

      insert into *table*1 **select** * **from** *table*1

  would cause problems)

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

    update *account*
    set *balance* = *balance* ∗ 1.06
    where *balance* > 10000

    update *account*
    set *balance* = *balance* ∗ 1.05
    where *balance* ≤ 10000
  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account
set balance =  case
                  when balance <= 10000 then balance *1.05
                  else   balance * 1.06
               end
```

Update of a View

- Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

  create view *loan_branch* **as**
  select *loan_number, branch_name*
  from *loan*

- Add a new tuple to *branch_loan*

  insert into *branch_loan*
  values ('L-37‘, 'Perryridge‘)

  This insertion must be represented by the insertion of the tuple

  ('L-37', 'Perryridge', *null*)

  into the *loan* relation

Updates Through Views (Cont.)

- Some updates through views are impossible to translate into updates on the database relations
  - **create view** *v* **as**

    select *loan_number, branch_name, amount*
    from *loan*
    where *branch_name* = ‘Perryridge’

  insert into *v* **values (**'L-99', 'Downtown', '23'**)**

- Others cannot be translated uniquely
  - **insert into** *all_customer* **values** ('Perryridge', 'John')
    - Have to choose loan or account, and create a new loan/account number!
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Joined Relations**

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Joined Relations – Datasets for Examples

- Relation *loan*
- Relation *borrower*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations – Examples

- *loan* **inner join** *borrower* **on**
  loan.loan_number = borrower.loan_number

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan* **left outer join** *borrower* **on**
  loan.loan_number = borrower.loan_number

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

- *loan* **natural inner join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*	*loan_number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan* **natural right outer join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

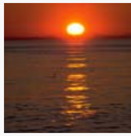
Joined Relations – Examples

- *loan* **full outer join** *borrower* **using** (*loan_number*)

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

- Find all customers who have either an account or a loan (but not both) at the bank.

  select *customer_name*
  from (*depositor* **natural full outer join** *borrower*)
  where *account_number* **is null or** *loan_number* **is null**



End of Chapter 3

Figure 3.1: Database Schema

branch (*branch_name*, *branch_city*, *assets*)

customer (*customer_name*, *customer_street*, *customer_city*)

loan (*loan_number*, *branch_name*, *amount*)

borrower (*customer_name*, *loan_number*)

account (*account_number*, *branch_name*, *balance*)

depositor (*customer_name*, *account_number*)

Figure 3.3: Tuples inserted into *loan* and *borrower*

loan_number	*branch_name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

loan

customer_name	*loan_number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	*null*

borrower

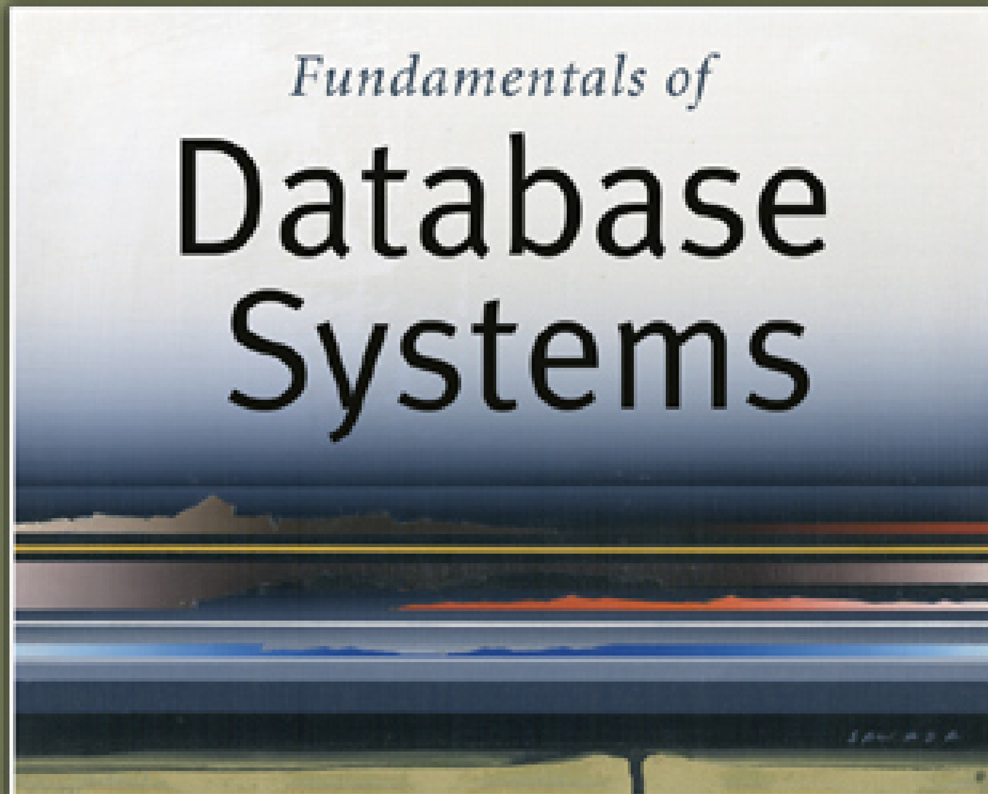
Figure 3.4:
The *loan* and *borrower* relations

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

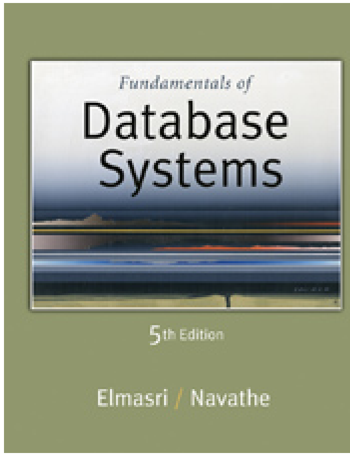


5th Edition

Elmasri / Navathe

Chapter 14

Indexing Structures for Files



Chapter Outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys

Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries **<field value, pointer to record>**, which is ordered by field value
- The index is called an access path on the field.

Indexes as Access Paths (contd.)

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values

Indexes as Access Paths (contd.)

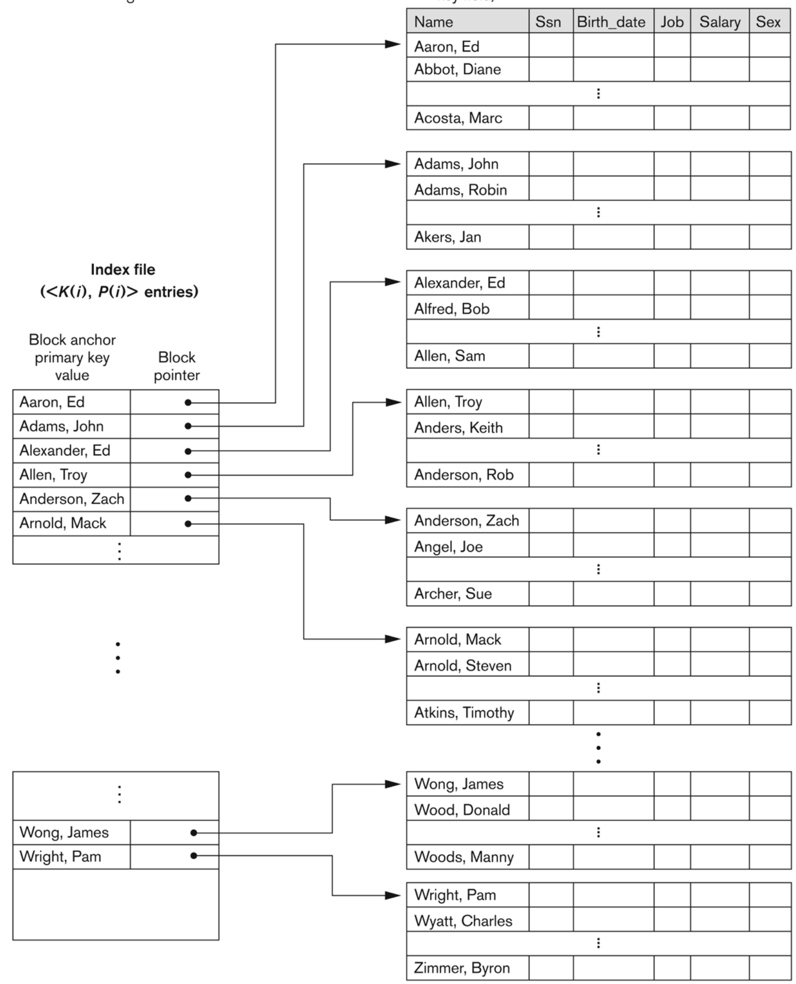
- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- Suppose that:
  - record size R=150 bytes block size B=512 bytes r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks
- For an index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the record pointer size P_R=7 bytes. Then:
  - index entry size $R_I=(V_{SSN}+P_R)=(9+7)=16$ bytes
  - index blocking factor Bfr_I= B div R_I= 512 div 16= 32 entries/block
  - number of index blocks b= (r/ Bfr_I)= (30000/32)= 938 blocks
  - binary search needs $\log_2 bI= \log_2 938$= 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 30000/2= 15000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b= \log_2 30000$= 15 block accesses

Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a **key field**
  - Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
  - A similar scheme can use the *last record* in a block.
  - A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

Primary index on the ordering key field

Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.

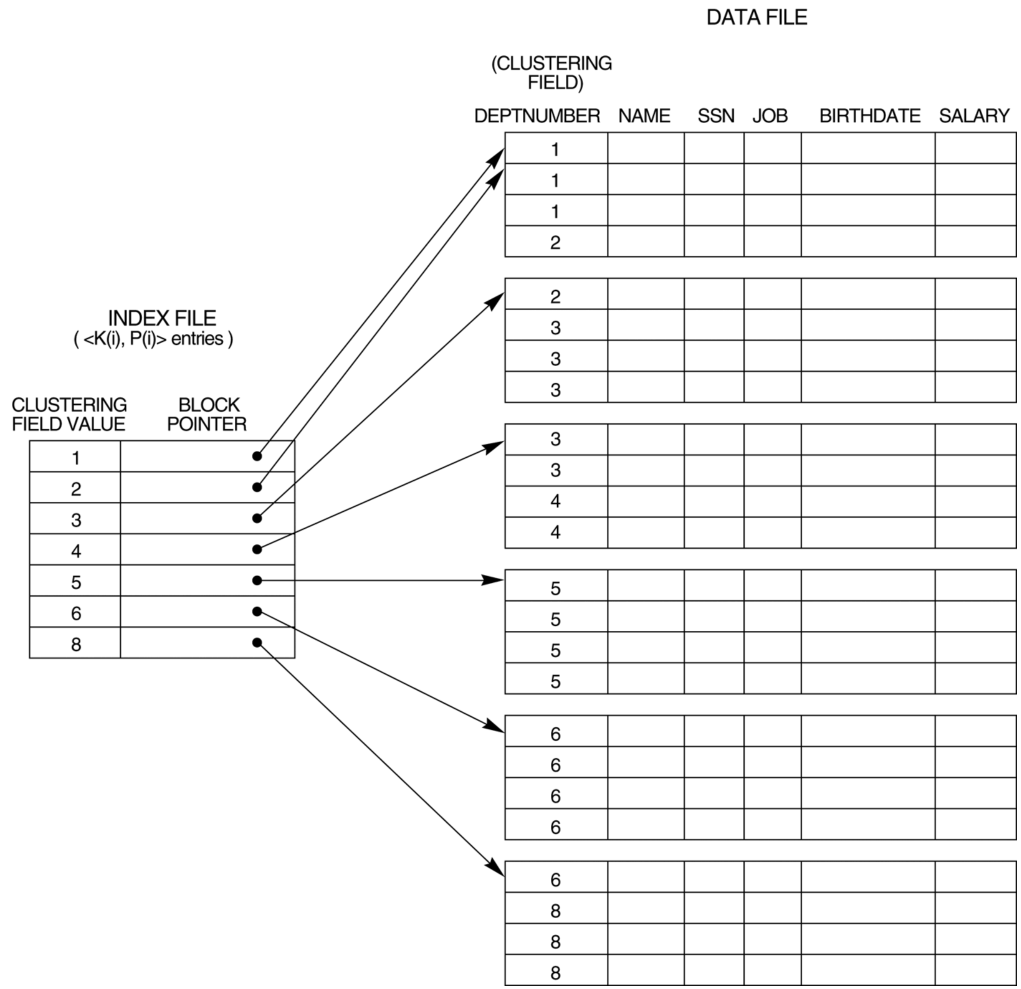


Types of Single-Level Indexes

- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.

A Clustering Index Example

- FIGURE 14.2
  A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file.



Another Clustering Index Example

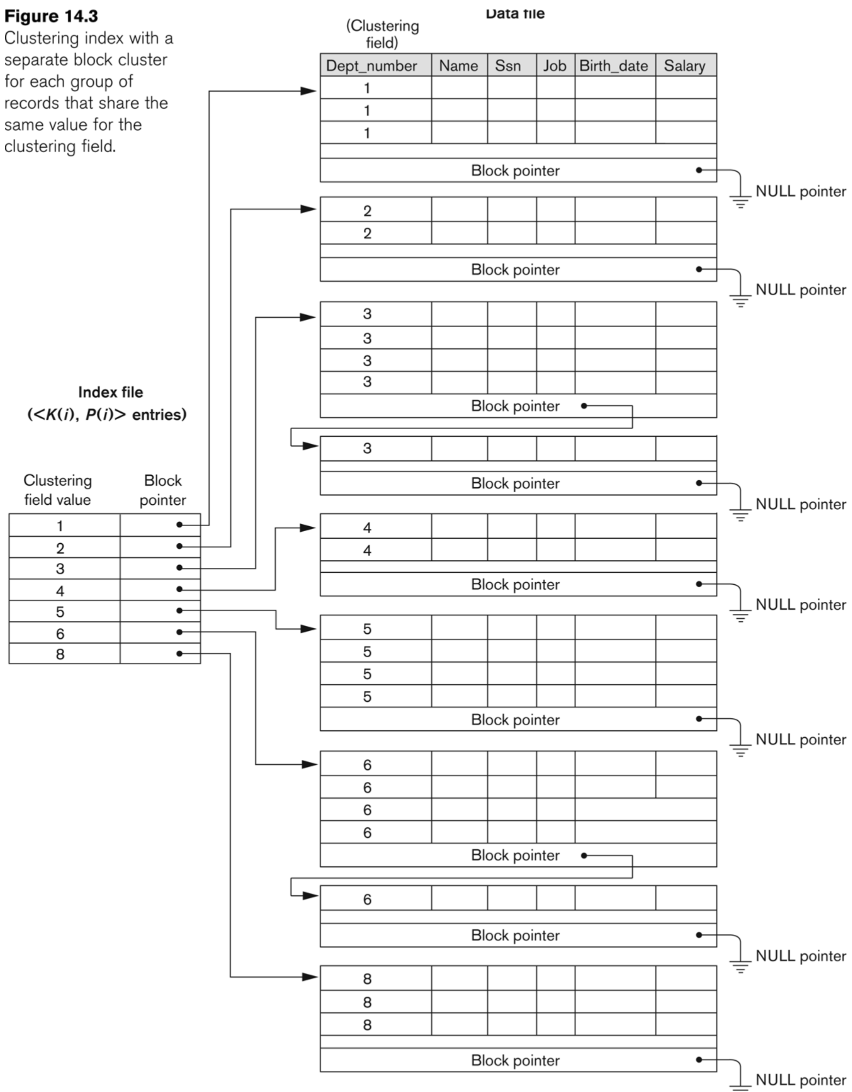


Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Types of Single-Level Indexes

- Secondary Index
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
    - The second field is either a **block** pointer or a record pointer.
    - There can be *many* secondary indexes (and hence, indexing fields) for the same file.
  - Includes one entry *for each record* in the data file; hence, it is a *dense index*

Example of a Dense Secondary Index

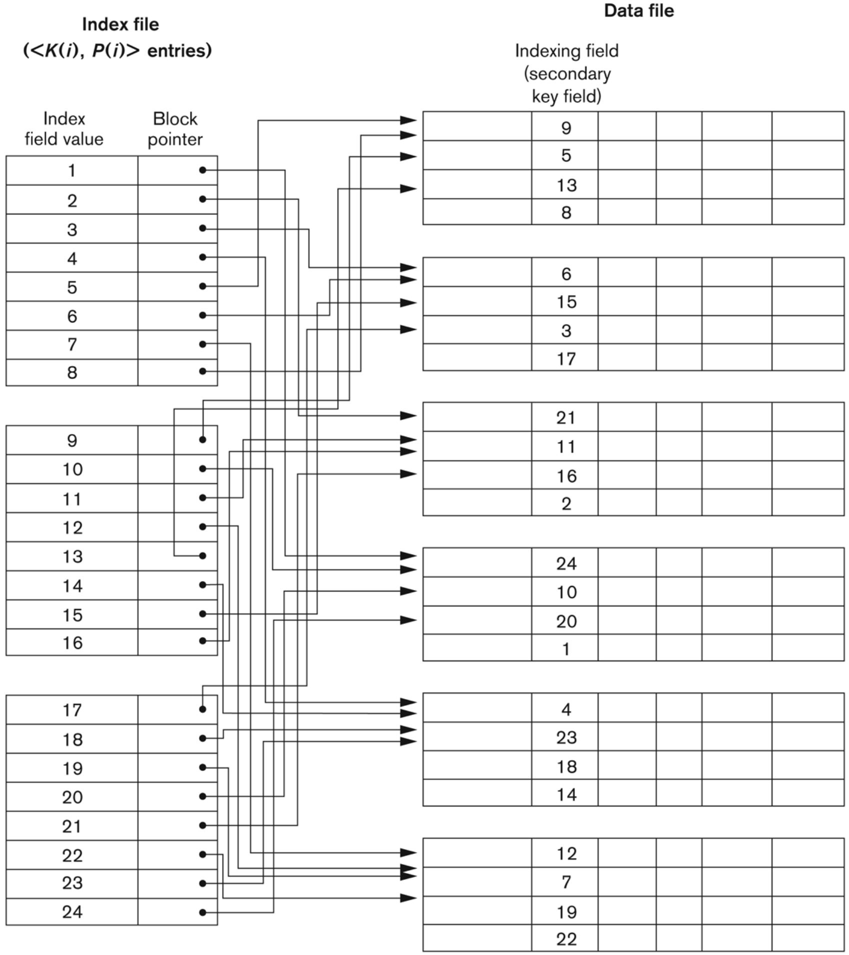


Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.

An Example of a Secondary Index

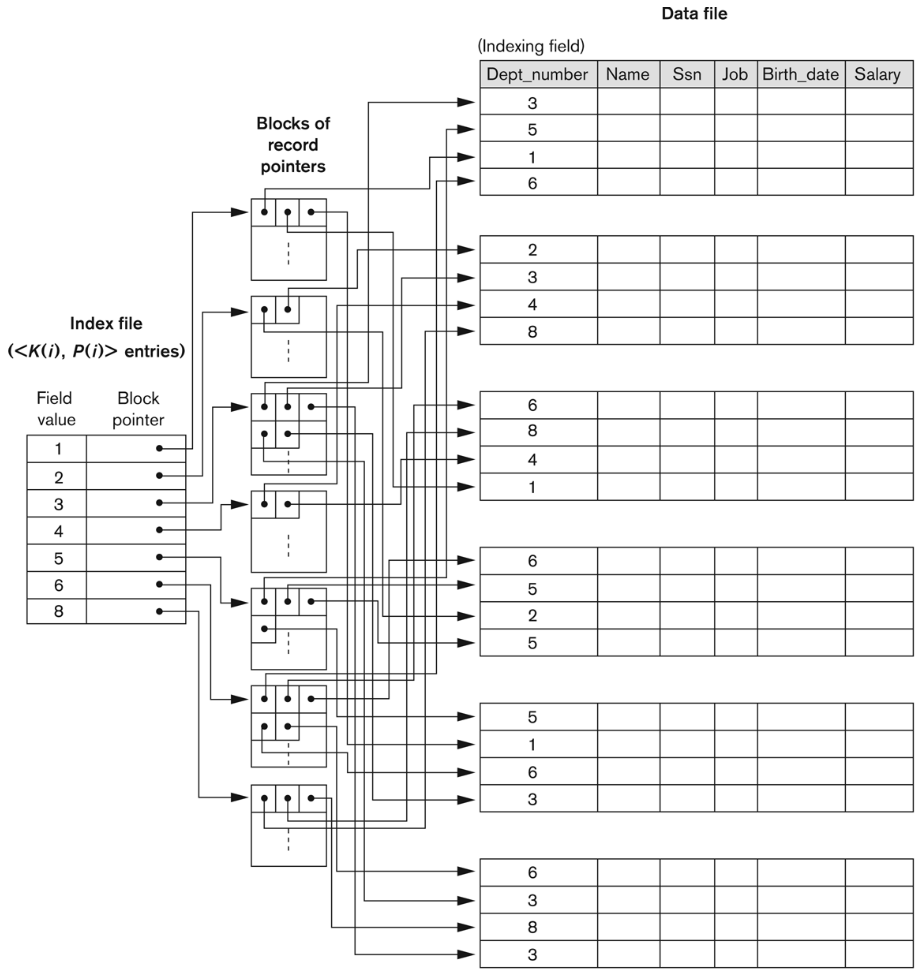


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

Properties of Index Types

TABLE 14.2 PROPERTIES OF INDEX TYPES

TYPE OF INDEX	NUMBER OF (FIRST-LEVEL) INDEX ENTRIES	DENSE OR NONDENSE	BLOCK ANCHORING ON THE DATA FILE
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no[a]
Secondary (key)	Number of records in data file	Dense	No
Secondary (nonkey)	Number of records[b] or Number of distinct index field values[c]	Dense or Nondense	No

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.

[b]For option 1.

[c]For options 2 and 3.

Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

A Two-level Primary Index

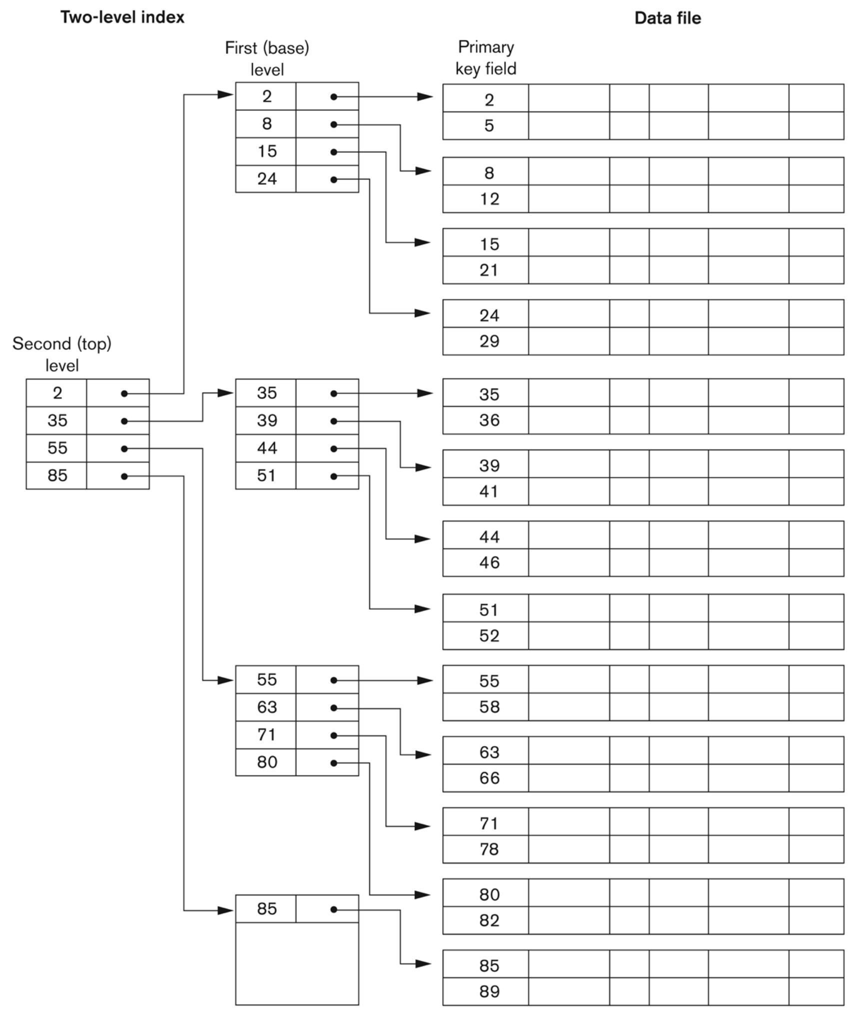


Figure 14.6
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

Multi-Level Indexes

- Such a multi-level index is a form of *search tree*
  - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

A Node in a Search Tree with Pointers to Subtrees below It

- FIGURE 14.8

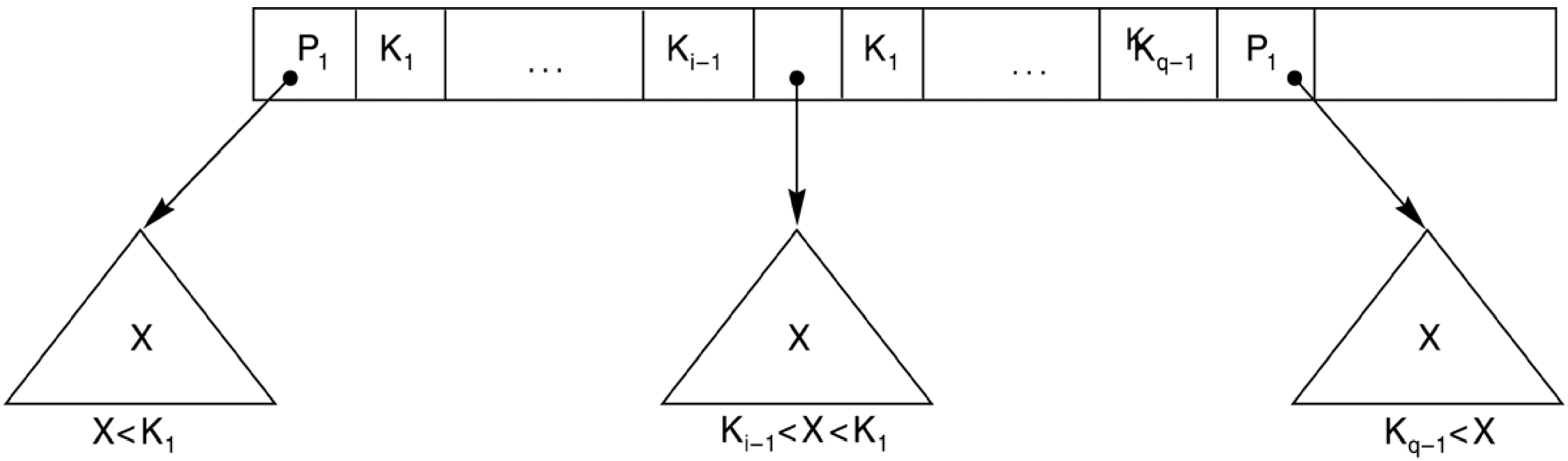
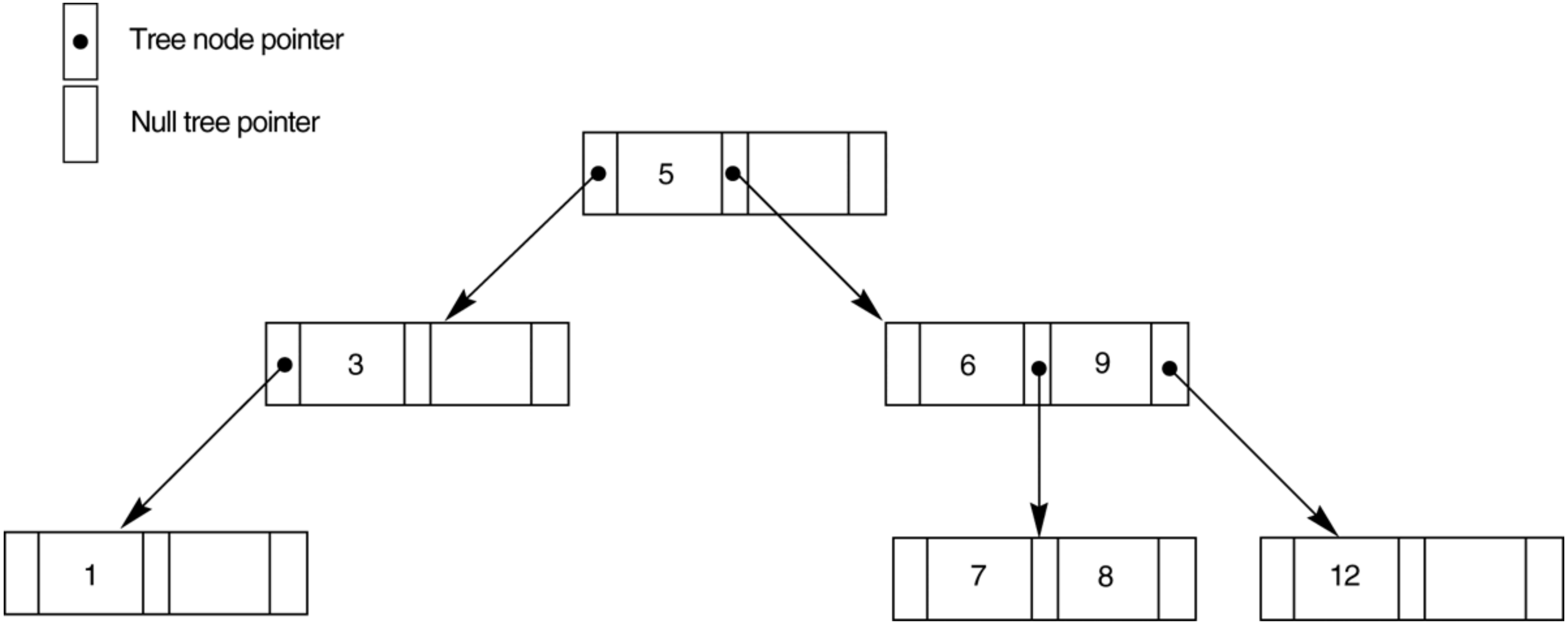


FIGURE 14.9
A search tree of order p = 3.



Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Most multi-level indexes use B-tree or B+-tree data structures because of the insertion and deletion problem
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B-Tree and B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

Dynamic Multilevel Indexes Using B-Trees and B+-Trees (contd.)

- An insertion into a node that is not full is quite efficient
  - If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree
- In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree

B-tree Structures

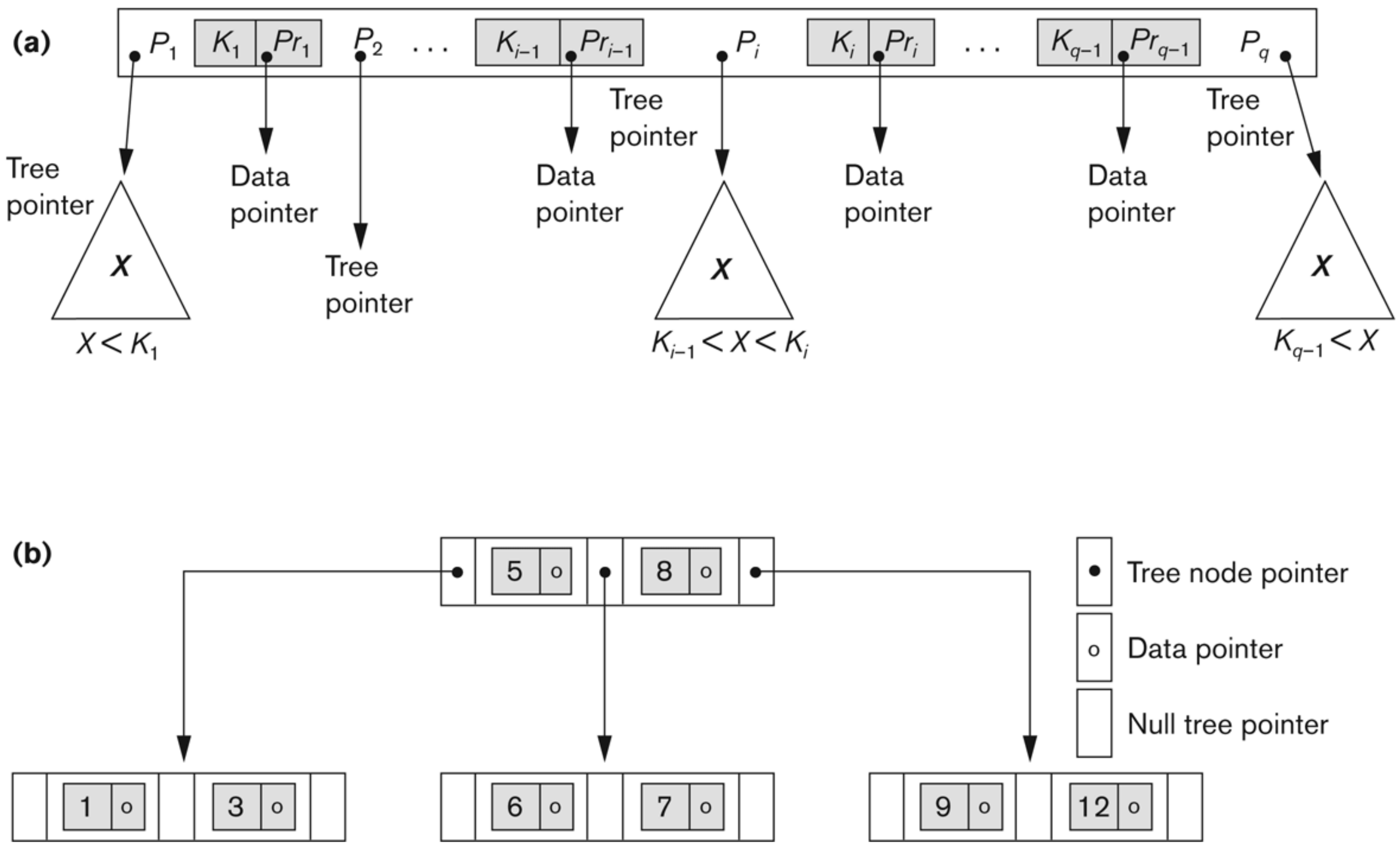
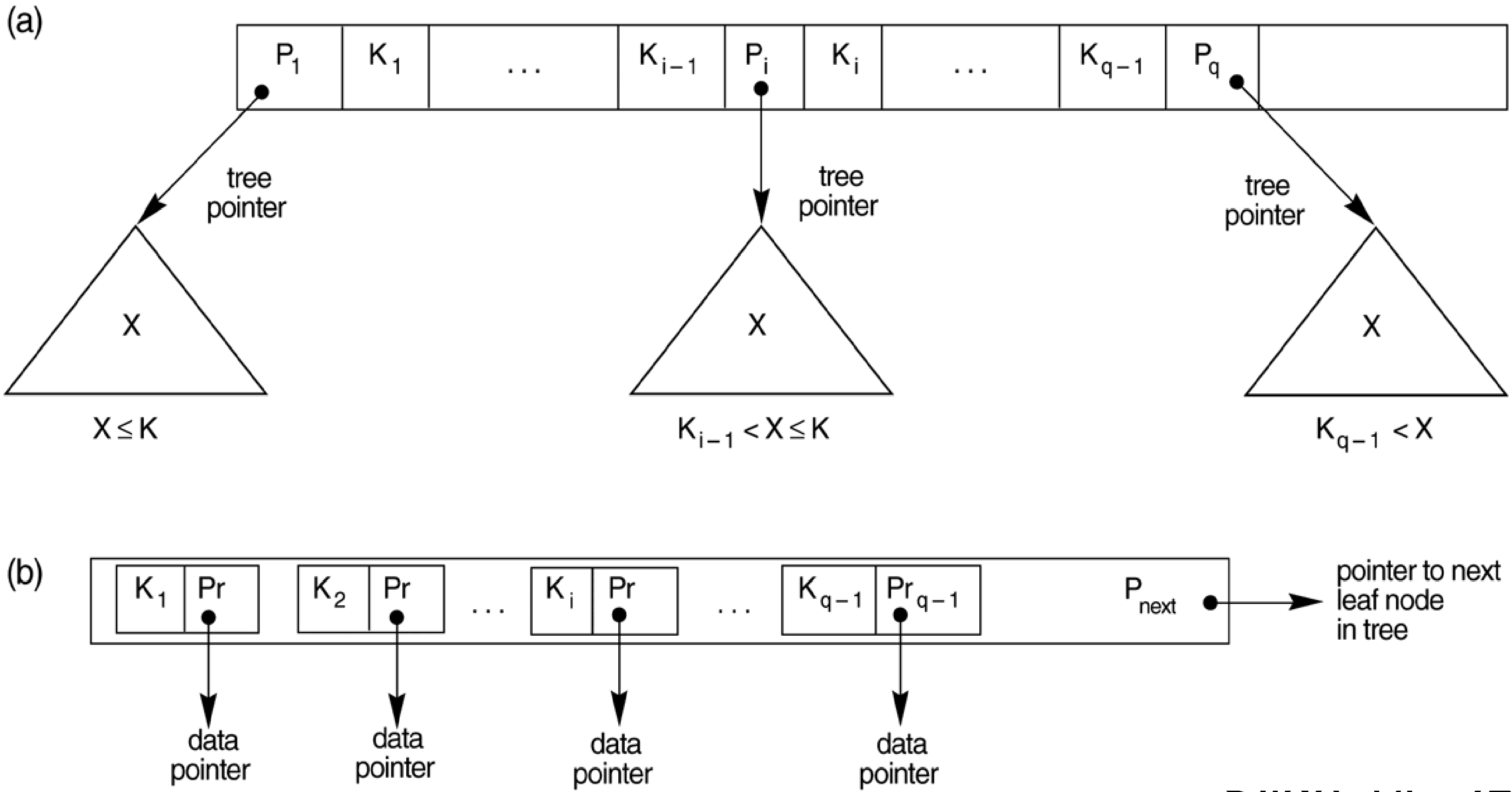


Figure 14.10
B-Tree structures. (a) A node in a B-tree with $q - 1$ search values. (b) A B-tree of order $p = 3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6.

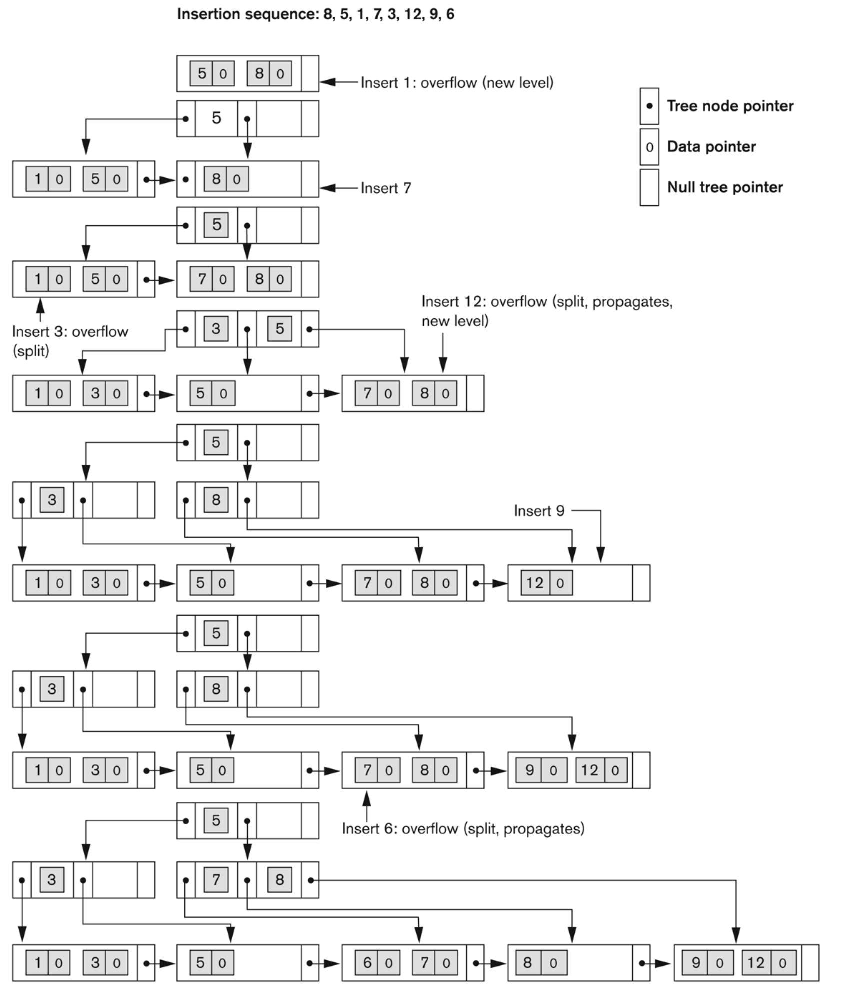
The Nodes of a B+-tree

- FIGURE 14.11 The nodes of a B+-tree
  - (a) Internal node of a B+-tree with q –1 search values.
  - (b) Leaf node of a B+-tree with q – 1 search values and q – 1 data pointers.



An Example of an Insertion in a B+-tree

Figure 14.12
An example of insertion in a B^+-tree with $p = 3$ and $p_{leaf} = 2$.



An Example of a Deletion in a B+-tree

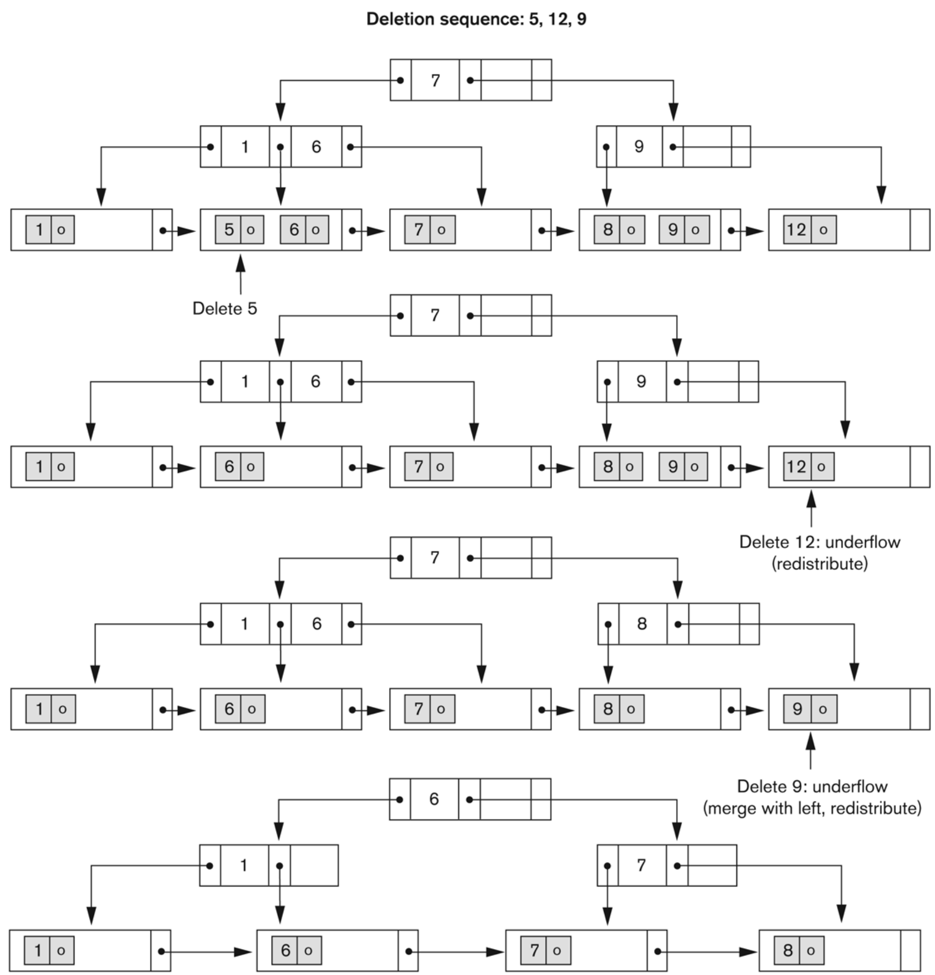


Figure 14.13
An example of deletion from a B+-tree.

Summary

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys

Normalization

Module 5

Ruchika Saini

Assistant Professor, DSU

Normalization

- Normalization process takes a relation schema through a series of tests to certify whether it satisfies a certain normal form.

- It can be considered as a process of analysing the given relation schemas based on their FDs (functional dependencies) and primary keys to achieve the desirable properties of –
  - Minimizing redundancy
  - Minimizing the anomalies
- The process of normalization through decomposition must also confirm the existence of additional properties that the relational schemas should process –
  - Lossless join property
  - Dependency preservation property

Types of normalization

- 1 NF
- 2 NF
- 3 NF
- BCNF

Definitions of Keys and Attributes Participating in Keys (1)

- Super Key
- Candidate Key
- Primary key
- Example – Book (isbn, title, author)

Attributes	Super Key	Candidate Key	Primary Key
isbn	Yes	Yes	Yes
isbn, title	Yes	No	No
title, author	Yes	Yes	Secondary key
title	No	No	No

Definitions of Keys and Attributes Participating in Keys (2)

- A **Prime attribute** must be a member of some candidate key
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

First Normal Form (1NF)

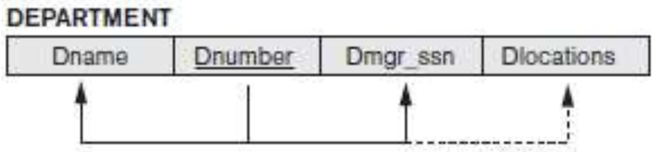
- Disallows
  - composite attributes
  - multivalued attributes
  - **nested relations**; attributes whose values for an *individual tuple* are non-atomic
- It states that domain of an attribute must include only atomic(simple and indivisible) values.

Normalization into 1NF

- There are three main techniques to achieve first normal form for such a relation:

1. Decomposing the non 1NF relation into two 1NF relations, take out the attribute that violates 1NF and the primary key and place it in a separate relation.
2. Expand the key so that there will be a separate tuple in the original relation for each value of a multivalued attribute.
3. If a maximum number of values is known for the attribute—for example, if it is known that at most three locations can exist for a department—replace the Dlocations attribute by three atomic attributes: Dlocation1, Dlocation2, and Dlocation3.

(a)



(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

Normalization into 1NF

- Disadvantages of applying 2^{nd} and 3^{rd} technique –
  - 2^{nd} technique disadvantage - it will introduce redundancy in the relation
  - 3^{rd} technique disadvantage – it will introduce NULL values if most departments have fewer than three locations

Normalization into 1NF

(a)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure

Normalization into 1NF.

(a) A relation schema that is not in 1NF.

(b) Sample state of relation DEPARTMENT.

(c) 1NF version of the same relation with redundancy.

Normalizing nested relations into 1NF

(a)

EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)

EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c)

EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Figure

Normalizing nested relations into 1NF.

(a) Schema of the EMP_PROJ relation with a nested relation attribute PROJS.

(b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple.

(c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key

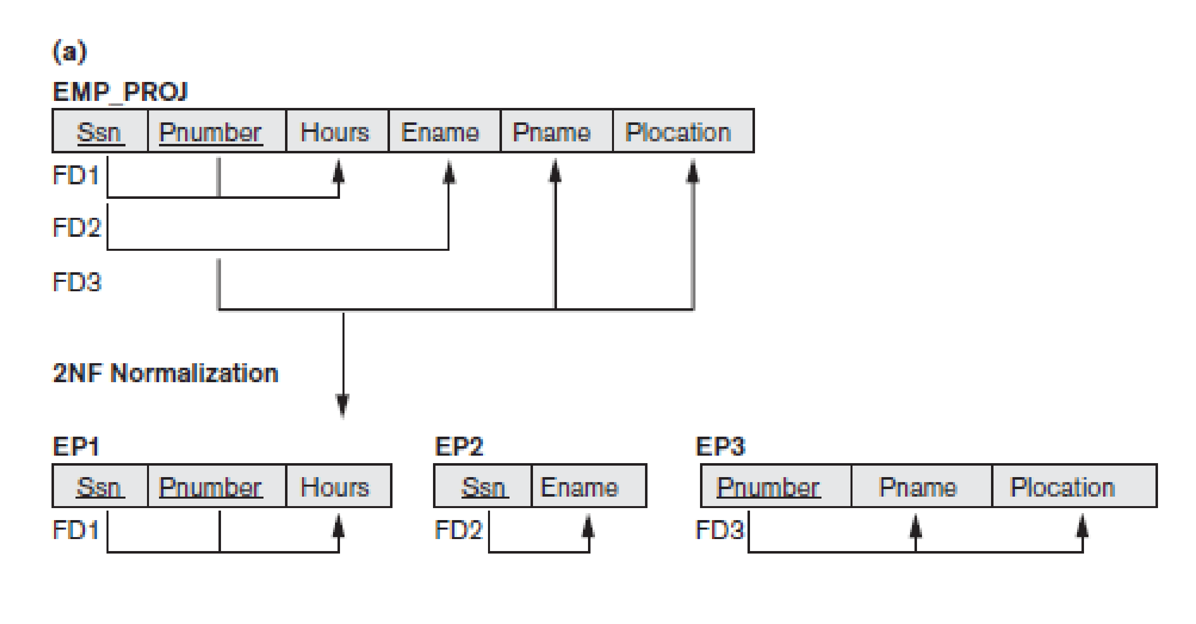
Second Normal Form (2NF)

- Uses the concepts of **FDs, primary key**
- Definitions
  - **Prime attribute:** An attribute that is member of the primary key K
  - **Full functional dependency:** a FD Y -> Z where removal of any attribute from Y means the FD does not hold any more (eg. SID, CID -> marks)
- Examples:
  - {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold
  - {SSN, PNUMBER} -> ENAME is not a full FD (it is called a partial dependency) since SSN -> ENAME also holds

2NF (continued)

- A relation schema R is in **second normal form (2NF)** if and only if it is in 1NF and every non-prime attribute A in R is fully functionally dependent on the primary key
- R can be decomposed into 2NF relations via the process of 2NF normalization or “second normalization”

Normalizing into 2NF



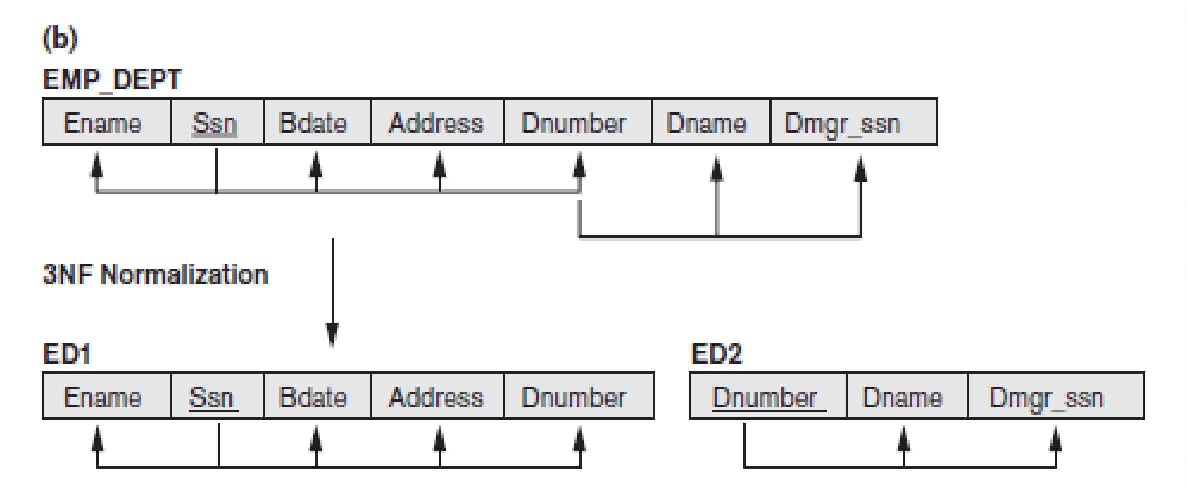
FDs –
Ssn, pno -> hours
Ssn -> ename
pno -> pname, plocation

Figure -
Normalizing into 2NF and 3NF.
(a) Normalizing EMP_PROJ into 2NF relations.

Third Normal Form (3NF)

- **Transitive functional dependency:** a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z
- A relation schema R is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key
- R can be decomposed into 3NF relations via the process of 3NF normalization
- NOTE:
  - In X -> Y and Y -> Z, with X as the primary key, we consider this a problem only if Y is not a candidate key.

Normalization into 3NF



FDs –

Ssn -> ename, bdate, address, dnumber

Dnumber -> Dmgr_ssn

Ssn -> Dmgr_ssn

Figure -

(b) Normalizing EMP_DEPT into 3NF relations.

Summary of all three normal forms

Normal Form	Test	Remedy (Normalization)
First (1NF)	Relation should have no multivalued attributes or nested relations.	Form new relations for each multivalued attribute or nested relation
Second (2NF)	For relations where primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key.	Decompose and set up a new relation for each partial key with its dependent attribute(s).Make sure to keep a relation with the original primary key and any attributes that are fully functionally dependent on it.
Third (3NF)	Relation should not have a nonkey attribute functionally determined by another nonkey attribute (or by a set of nonkey attributes). That is, there should be no transitive dependency of a nonkey attribute on the primary key.	Decompose and set up a relation that includes the nonkey attribute(s) that functionally determine(s) other nonkey attribute(s).

Boyce-Codd normal form

- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an **FD X → A** holds in R, then **X is a superkey** of R
- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF
  - Every 3NF relation is in 2NF
  - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- Hence BCNF is considered a stronger form of 3NF
- The goal is to have each relation in BCNF (or 3NF)

Normalization into BCNF

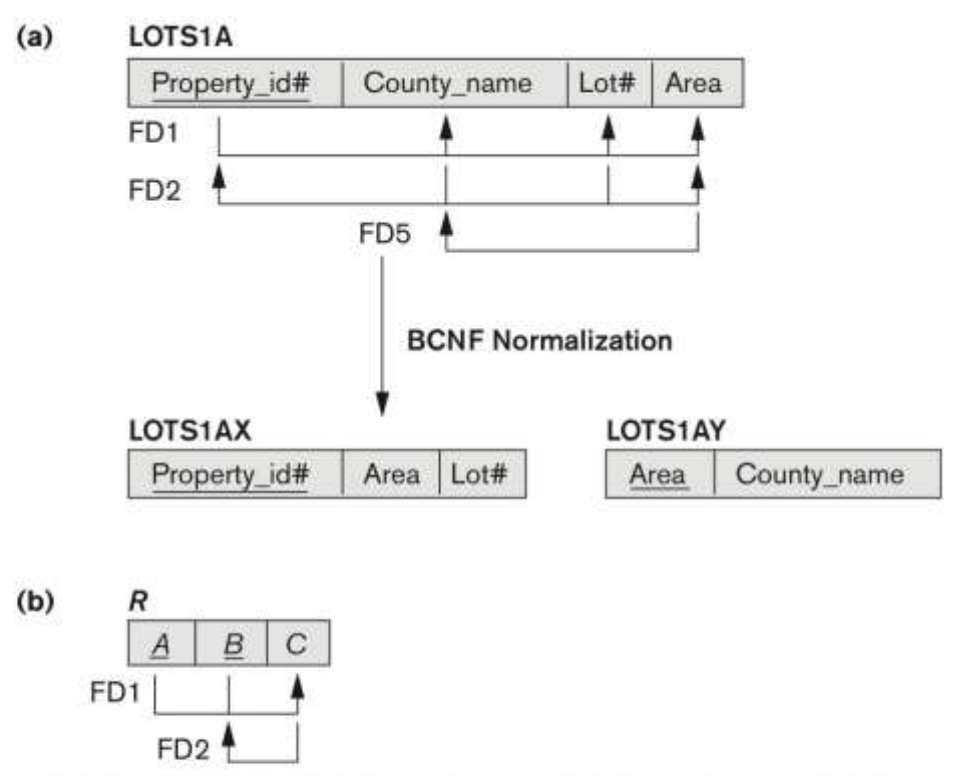


Figure
Boyce-Codd normal form.
(a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition.
(b) A schematic relation with FDs; it is in 3NF, but not in BCNF due to the f.d. C → B.

Example

Q. Let us assume a table User_Personal as given below:

User_Personal table holds the following set of functional dependency: -

- UserID → email Name City State Zip Phone no
- Zip → City State

i. Is this table in First Normal Form? If yes then why? And if not then convert into 1NF.

ii. Is this table in Second Normal Form? If yes then why? And if not then convert into 2NF.

iii. Is this table in 3NF? If yes then why? And if not then convert into 3NF.

User id	email	Name	City	State	Zip	Phone no
MA12	mani@gmail.com	Manish	Vadodara	Gujrat	832212	67839, 64889
PO45	pooja@gmail.com	Pooja	mumbai	Maharashtra	436729	45379, 98657
LA33	lav@gmail.com	Lavleen	Bangalore	Karnataka	562937	61527, 89756, 97578
DA7	dany@gmail.com	Dany	Trichy	Tamilnadu	787579	43251