CS157A: Introduction to Database Management Systems

Chapter 8: Views and Indexes

Suneuy Kim

Views

- Virtual view
  - A relation that is the result of a query over other relations.
  - Virtual views are not stored in the databases
- Materialized views are periodically constructed from the database and stored there.
  - Enable efficient access to the database
  - Are most useful in data warehousing scenarios, where frequent queries of the actual base tables can be extremely expensive.

Declaring Views

```
DROP VIEW IF EXISTS SeniorUsers;
CREATE VIEW SeniorUsers AS
  SELECT uID, uName, age
  FROM User
  WHERE age >=60;
```

Querying Views

- A view is queried as if it was a stored table.
- The query processor replaces the view by its definition to process the query.


```
SELECT *
FROM SeniorUsers, Loan
WHERE SeniorUsers.uID = Loan.uID and overdue = 1;
```



```
SELECT *
FROM (SELECT uID, uName, age FROM User
      WHERE age >=60) SU, Loan
WHERE SU.uID = Loan.uID and overdue = 1;
```

View Removal

- DROP VIEW SeniorUsers;
  - Will not affect the base table User.
  - Can't do any query involving SeniorUsers.
- DROP TABLE User;
  - Will remove the table User from the database and also make all views referring the User table unusable.

Modifying Views

- We can't modify (insert, delete, update) a view like a table since a view is not stored.
- However, for the users who access the database only though views, there should be a way to modify views – modification to a view should be reflected on the base tables.
- Rewrite a query that modifies a view in a way that it modifies the base tables.

Example

```
DROP VIEW IF EXISTS YoungUsers;
CREATE VIEW YoungUsers AS
  SELECT uID, age
  FROM User
  WHERE age < 19;
```

YoungUsers

VIEW

uID	age
12	17
24	10

Insert Into YoungUsers
Values (77, 17);

User

Base Table

How can we translate this query to modify the database ?

Insert into User Values (77, ?, 17); // so many different ways !

Modifying Views

- Correctness of translation can be achieved but ambiguity issue should be resolved and it is not straightforward for some cases.

Two main approaches of translating modification to a view

1. INSTEAD-OF Triggers on Views

   e.g. SQLite, Postgres, Oracle

2. Automatically done by DBMS

   e.g. MySQL

INSTEAD OF Triggers on Views

- INSTEAND OF in place of BEFORE or AFTER.
- When an event occurs, the action of the trigger is done instead of the event.
- Translation is put in the action part of the trigger.
- All modifications can be handled.
- No guarantee of correctness

Automatic translation done by DBMS

- Translation to base table is automated by restricting views and modifications.
- No translation needs to be done by SQL programmers.
- Restrictions are significant.
- Adopted by the SQL standard

Example: Modifying a View

```
DROP VIEW if exists LateFeeUsers;
CREATE VIEW LateFeeUsers AS
SELECT uID, title, loanDate, overdue
FROM LOAN
WHERE overdue =1;

DELETE FROM LateFeeUsers WHERE uID = 135;
```


SQLite complains that "cannot modify LateFeeUsers because it is a view."

Instead of Triggers: Delete

```
Drop trigger LateFeeUsersDelete;
Create Trigger LateFeeUsersDelete
instead of delete on LateFeeUsers
for each row
begin
  delete from LOAN
  where uID = OLD.uID and overdue = 1;
end;

delete from LateFeeUsers
where uID = 135;
```


- For each deleted row, the action is taken.
- The deleted row means a tuple that will be logically deleted from the view.
- The OLD variable is the tuple that Is asked to be delete from the view.
- Result: The user 135 is deleted from both LOAN and the view.

Instead of Triggers: Update

```
DROP Trigger LateFeeUsersUpdate;
CREATE Trigger LateFeeUsersUpdate
instead of update of title
ON LateFeeUsers
for each row
begin
  update LOAN
  set title=New.title
  where uID = Old.uID and title =
Old.title AND overdue = 1;
end;
```



```
update LateFeeUsers
set title = 'Bambi II'
WHERE uID = 24 AND
title = 'Bambi';
```

Result: The title of uID 24 with ‘Bambi II’ in both Loan and the view.

Instead of Trigger: Insert

```
DROP trigger LateFeeUsersInsert;
CREATE trigger LateFeeUsersInsert
instead of INSERT ON LateFeeUsers
for each row
begin
   INSERT INTO LOAN VALUES
  (New.uID, New.title,
New.loanDate,New.overdue);
END;
```



```
INSERT INTO
LateFeeUsers
VALUES (888,
'Gone With the
Wind', '2000-12-
25',1) ;
```

Result: The new tuple is inserted in both Loan and the view.

Wrong Translations

- DBMS can't prevent an incorrect trigger (written with logical errors) from being triggered!

Example: Wrong Translations-delete

```
Drop trigger LateFeeUsersDelete;
Drop trigger WrongTrigger;

Create Trigger WrongTrigger
instead of delete on LateFeeUsers
for each row
begin
  delete from LOAN
  where uID = OLD.uID and overdue = 1
AND title = 'somebook' ;
end;

delete from LateFeeUsers
where uID =456;
```

Result: There is no change in the Loan Table and the View.

Example: Wrong Translations-update

```
DROP Trigger LateFeeUsersUpdate;
Drop Trigger WrongTrigger;

CREATE Trigger WrongTrigger
instead of update of title ON LateFeeUsers
for each row
begin
update LOAN
  set title=New.title
  where uID = Old.uID and title = Old.title AND
loanDate >= datetime('2013-01-01 00:00:00');
end;

update LateFeeUsers
set title = 'Bambi II'
WHERE uID = 234 AND title = 'Bambi';
```


- Result: There is no change in the view, and a wrong tuple is updated in the Loan table.
- Reason: overdue = 1 is missing and a wrong condition is added in the where clause.

Example: Wrong Translation - insert

- With the LateFeeUsersInsert trigger, do

  INSERT INTO LateFeeUsers VALUES
  (**111**, 'Modern Database Systems', date('now'), **0**);

- Result: The tuple is inserted into the Loan table but doesn't show up in the view because it doesn't satisfy the condition of the view.
- User should not write an insert to a **view** that will change the base table but doesn't get reflected on the view.

Better LateFeeUsersInsert Trigger

```
DROP trigger LateFeeUsersInsert;
CREATE trigger LateFeeUsersInsert
instead of INSERT ON LateFeeUsers
for each row
WHEN New.overdue = 1
begin
 INSERT INTO LOAN VALUES
(New.uID, New.title, New.loanDate,New.overdue);
end;
```


INSERT INTO LateFeeUsers VALUES(**111**, 'Streaming Media', date('now'), **0**);

Result: No change in both Loan and the View !

View involving Joins

```
DROP VIEW IF EXISTS UsersBeingOverdue;
CREATE View UsersBeingOverdue as
  SELECT USER.uID, USER.age, title, overdue
  FROM USER,LOAN
  WHERE USER.uID = LOAN.uID AND overdue = 1;
```

View involving Joins: Insert

```
DROP trigger UsersBeingOverdueInsert;
CREATE trigger UsersBeingOverdueInsert
instead of INSERT ON UsersBeingOverdue
for each row
WHEN ((New.age = (select age from User where uID = New.uID)) and  New.title in
(select title from book) and New.overdue = 1)
begin
INSERT INTO LOAN VALUES (NEW.uID, New.title, NULL, New.overdue);
end;
```


- INSERT INTO UsersBeingOverdue VALUES(777, 30,'Lion King', 1); // will not add if 777 with age 30 is not in the user relation
- INSERT INTO UsersBeingOverdue // The library wouldn't do this in practice ☺

```
SELECT USER.uID, USER.age, 'Promise', 1
FROM USER
WHERE USER.uID NOT IN (SELECT uID FROM LOAN);
```

View involving Joins: delete

```
DROP Trigger UsersBeingOverdueDelete;
CREATE Trigger UsersBeingOverdueDelete
  instead of delete ON UsersBeingOverdue
  for each row
  begin
    delete FROM LOAN
    WHERE uID = Old.uID AND overdue = 1;
  end;

DELETE FROM  UsersBeingOverdue WHERE uID = 135;
```

View and Constraints

```
CREATE VIEW SeniorUsers AS
   SELECT uID, uName, age
   FROM User
   WHERE age >=60;
```



```
DROP TRIGGER IF EXISTS SeniorUsersInsert;
CREATE Trigger SeniorUsersInsert
  instead of insert ON SeniorUsers for each row
  when New.age >=60
 begin
INSERT INTO User VALUES(New.uID, New.uName, New.age, NULL);
  end;
```


- INSERT INTO SeniorUsers VALUES(**135**,'Kim', **70**); //error if 135 already exists in the User.

Ambiguous Translations

- Fundamentally, we could write a trigger for such a modification, but it does not make much sense to write a translation for this type of modification on the view.

Not-updatable view due to ambiguous translations

- View involving aggregation

```
DROP VIEW IF EXISTS AvgAge;
CREATE VIEW AvgAge AS
SELECT uName, AVG(age) as avg
FROM User
GROUP BY uName;
```


- Consider the following query:

```
update AvgAge set avg = 50 where uName = 'Kim';
```

Not-updatable view due to ambiguous translations

- View using distinct

```
DROP VIEW IF EXISTS UserNames;

CREATE VIEW UserNames AS
    SELECT DISTINCT uName FROM User;
```


- Consider the following query:

```
Insert into UserNames values ('Kim');
```

Not-updatable view due to ambiguous translations

- View where the sub query references the same table of outer query

```
CREATE VIEW UserWithSameAge AS
  SELECT * FROM USER U1
  WHERE EXISTS (SELECT * FROM USER U2
        WHERE  U1.uID <> U2.uID AND U1.age =
        U2.age);
```


- Consider the following query:

```
delete from UserWithSameAge where uID =1006;
```

- Restrictions in SQL Standard for "updatable views"
  - SELECT (no DISTINCT) from a single table R (can't be a join view)
  - Attributes not in SELECT can be NULL or can have a default value.
  - Where clause of a sub query must not involve R in the from clause of outer query (but allow to refer to other tables)
  - No group by or aggregation

Automatic View Modifications

```
DROP VIEW IF EXISTS LateFeeUsers;
CREATE VIEW LateFeeUsers AS
SELECT uID, title, loanDate, overdue
FROM LOAN WHERE overdue =1;
```



```
INSERT INTO LateFeeUsers VALUES (1019,
'Bambi','2000-12-25' , 1); // inserted in both
Loan and View without any trigger.
```

WITH CHECK OPTION - MySQL

```
DROP VIEW LateFeeUsers;
CREATE VIEW LateFeeUsers AS
SELECT uID, title, loanDate, overdue FROM LOAN
WHERE overdue =1 WITH CHECK OPTION;
```


- To prevent inserts to rows for which the WHERE clause in the select_statement is not true.

INSERT INTO LateFeeUsers VALUES (1018, 'Bambi','2013-12-25' , 0);

will CHECK OPTION FAIL.

- It also prevents visible rows from being updated to become nonvisible rows.

UPDATE LateFeeUsers SET overdue = 0 WHERE uID = 1001;

will CHECK OPTION FAIL.

View involving a subquery that refers to another table

```
DROP VIEW IF EXISTS YoungLoaners;
Create VIEW YoungLoaners AS
   SELECT uID, uName, age
   FROM User
   WHERE age < 18 AND User.uID in (select uID from Loan) WITH
CHECK OPTION ;

delete FROM YoungLoaners WHERE uID =1001;
```


- If the outer query refers a single table, which is required by standard, deletion will be done in the table of the outer query. (Thus, 1001 will be removed from User not from Loan)
- If this deletion from the user violates a foreign key constraint, it will be rejected or handled according to a given policy.

View involving a subquery that refers to another table

- Without the CHECK option, consider

INSERT INTO YoungLoaners VALUES (187, 'Wu', **60**);

- Without CHECK option: Wu will be inserted to User in the outer query but will not be shown in the view due to his age. This insertion to the user through the view YoungLoaners is not desirable.

- Solution: check with option. This insertion will be failed by the check option.

Join Views

- SQL Standard doesn't allow to update Join Views.
- MySQL allows insert and update operations on join views, but not delete operations.

Why Join Views are not updatable by standard ?

```
CREATE VIEW MovieProd AS
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```



```
Insert into MovieProd Values ('Bambi', 1969, 'Max');
```

Translation:

```
Insert into Movies Values ('Bambi', 1969, null);
Insert into MovieExec ('Max', null);
```

MySQL: Join Views

```
Drop View if EXISTS BambiUsers;
Create view BambiUsers(uID, loanID, uName, overdue) AS
  Select User.uID, Loan.uID, User.uName, overdue
  From User, Loan
  Where User.uID = Loan.uID and title = 'Bambi' ;
```


- update BambiUsers set overdue = **0** WHERE overdue = **1**;
  The system will modify the base table containing the attribute to be updated. (Loan in this case.)

- update BambiUsers set loanID = **1007** WHERE loanID = **1011**;
  Suppose there is a tuple with uID= 1007 in both User and Loan, this will be accepted. To prevent this update, use the check option.
- update BambiUsers set loanID = **111** WHERE loanID = **1011**;
  - Without check option, the uID in Loan is updated, but the uID in User is not changed, resulting in foreign key constrain violation.
  - With check option, CHECK OPTION FAILED.

INDEX

Indexes

- Index on an attribute A: A data structure that makes it faster to find those tuples that have specific column values
- A data structure of (key, value)
  - Key x: A value of attribute A → index key
  - Value: set of locations of the tuples that have x for the value of A.
- Index key can be any attribute or set of attributes
- Stored in database
- Underlying data structures
  - B tree/B+ tree
  - Hash table

B-tree

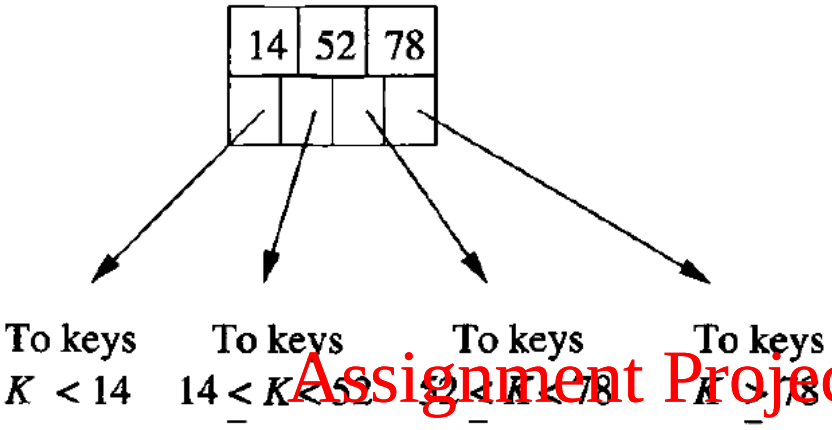


Figure 14.12: A typical interior node of a B-tree

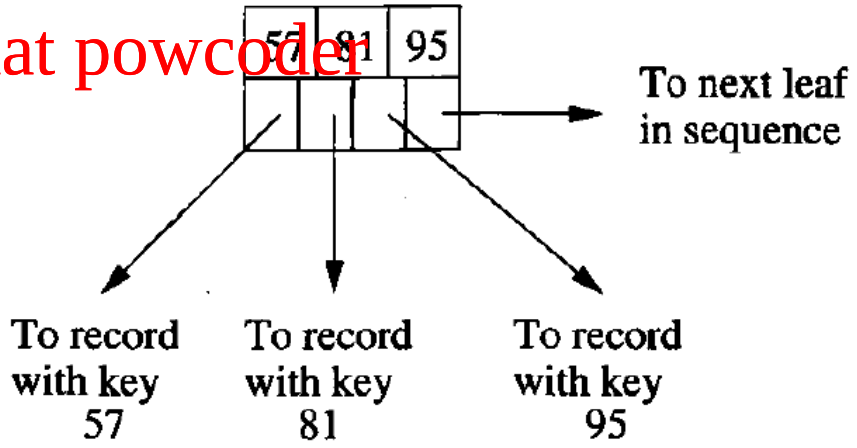


Figure 14.11: A typical leaf of a B-tree

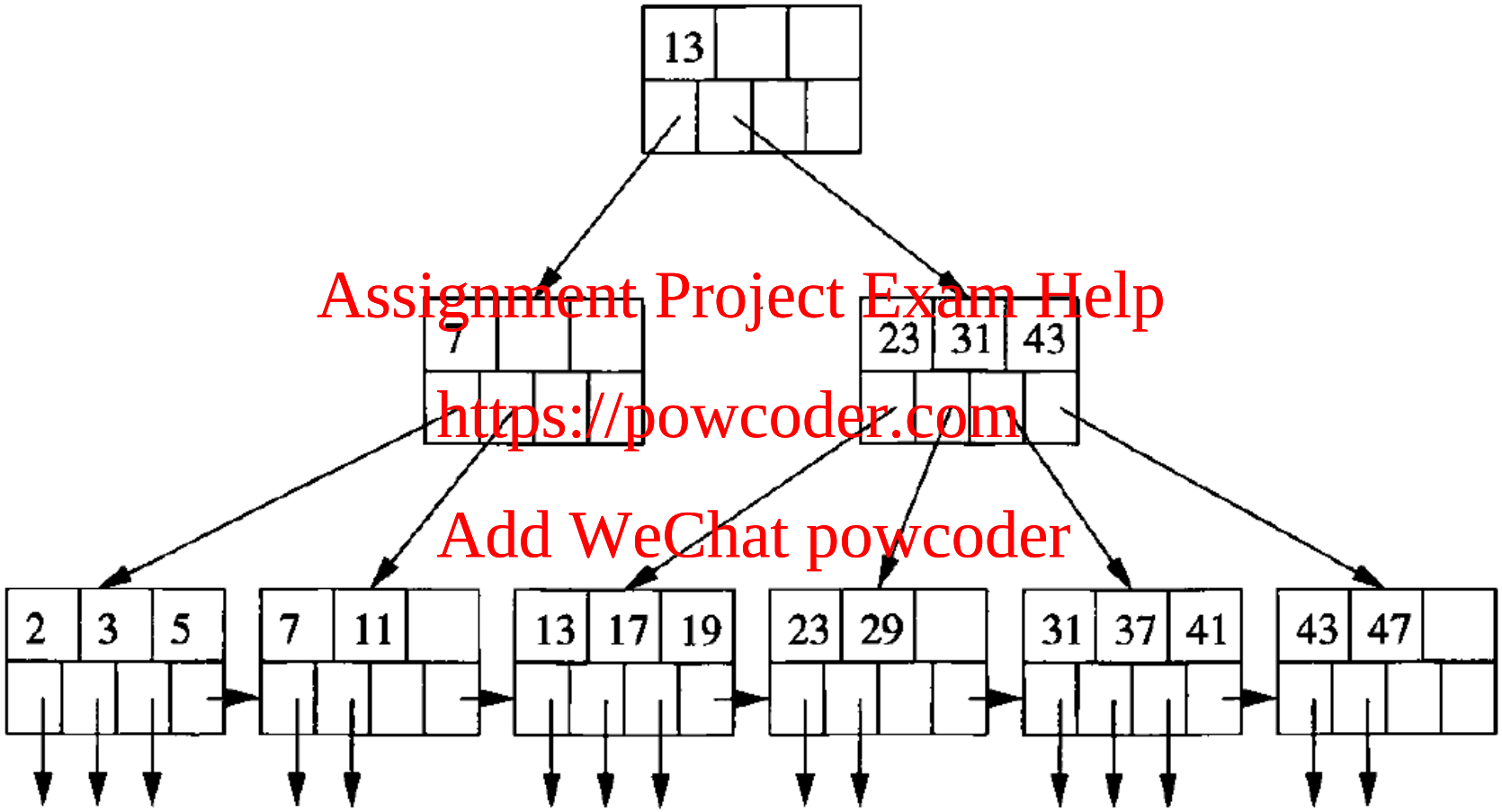


Figure 14.13: A B-tree

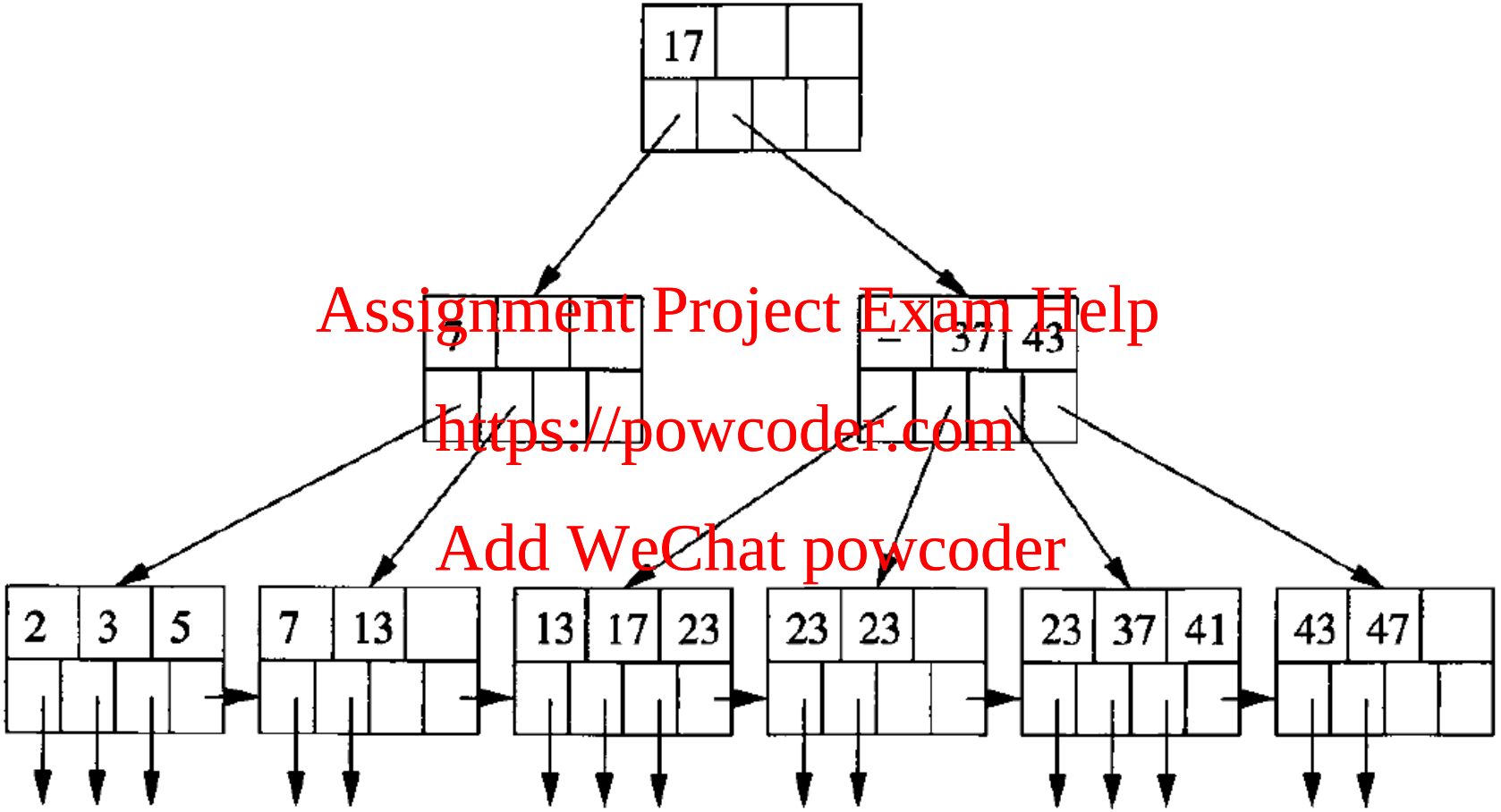


Figure 14.14: A B-tree with duplicate keys

Motivation

```
SELECT User.uID
FROM User INNER JOIN Loan
ON User.uID = Loan.uID
WHERE title = 'Bambi';
```


If we have an index on title of Loan, first get the tuples containing Bambi, and test uIDs of the tuples for equality.

Defining Indexes

CREATE INDEX TitleIndex ON Loan (title);

To serve the query in the previous slide, the DBMS can quickly determine the position of tuples that contain 'Barnhi' without having to look at all the data.

A single index on multiple attributes (Compound Index)

- Suppose title and year form a primary key for Movie. Then consider

```
SELECT *
FROM Movie
Where title = 'Star Wars' and year = 1990;
```


- With `CREATE INDEX keyIndex Movie(title,year)`

  The index will find only one tuple because the title and year together is the primary key of the Movie table.
- With `CREATE INDEX YearIndex Movie(year)`

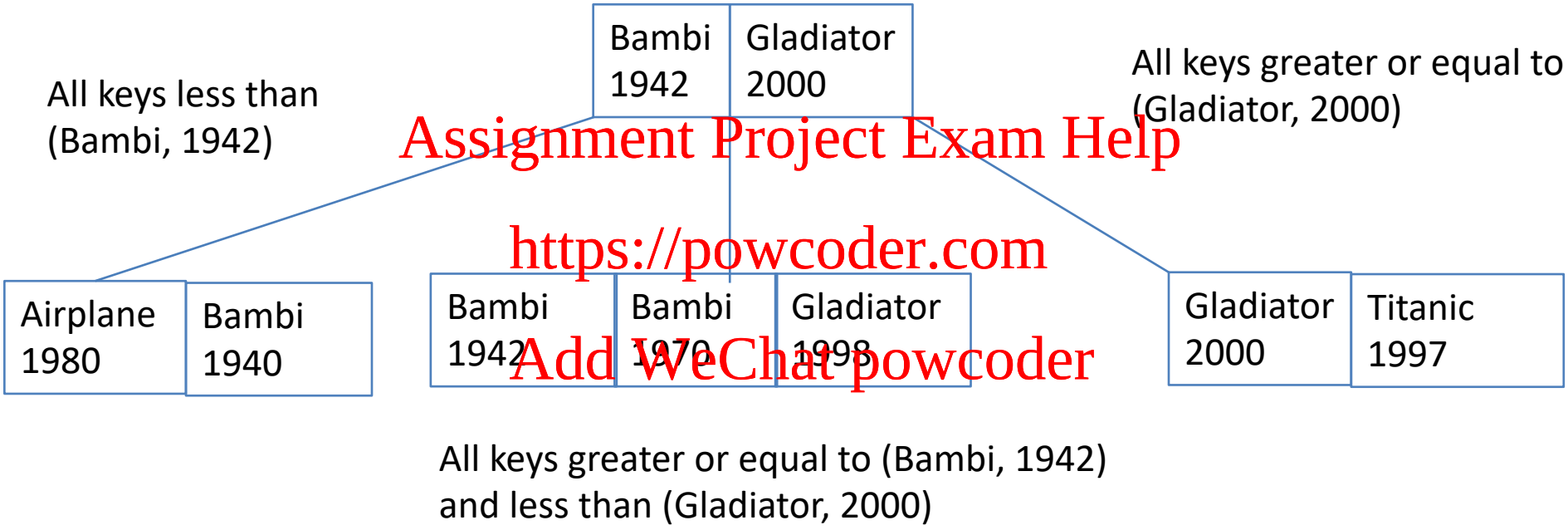
  First find all movies made on 1990 using the index and check through them for the given title.

Compound Index Key

- The ordering of the compound index key values is lexicographic ordering. That is, (a1, a2) < (b1, b2) if either a1 < b1 or (a1=b1 and a2 < b2)
- The order of the individual attributes in the compound index key is important.

`(title, year)` vs. `(year, title)`

Compound Index Key



The above index may be used to search for (title, year) or (title),but not for (year).

Selection of Indexes

- Benefits: faster query execution
- Costs: space, index creation, maintenance
  - Maintenance is the most expensive overhead: Insertions, deletions and updates to that relation become more complex and time-consuming.

Selection of Indexes

- The most useful index on a relation is an index on its key (key index)
  - Queries involving the key are very common.
  - At most one disk page needs to be retrieved because a key is associated with at most one tuple.
- There are two situations where an index can be effective, even if it is not on a key:
  - If the index key is almost a key (e.g. title instead of the key (title, year) for Movies) Even if each of the tuples with a given value is on a different page, we shall not have to retrieve many pages from the disk.
  - If the tuples are clustered on the index key

  We *cluster* a relation on an attribute by grouping the tuples with a common value for that attribute onto as few pages as possible. Then, even if there are many tuples, we shall not have to retrieve nearly as many pages as there are tuples.

Example: Selection of Index

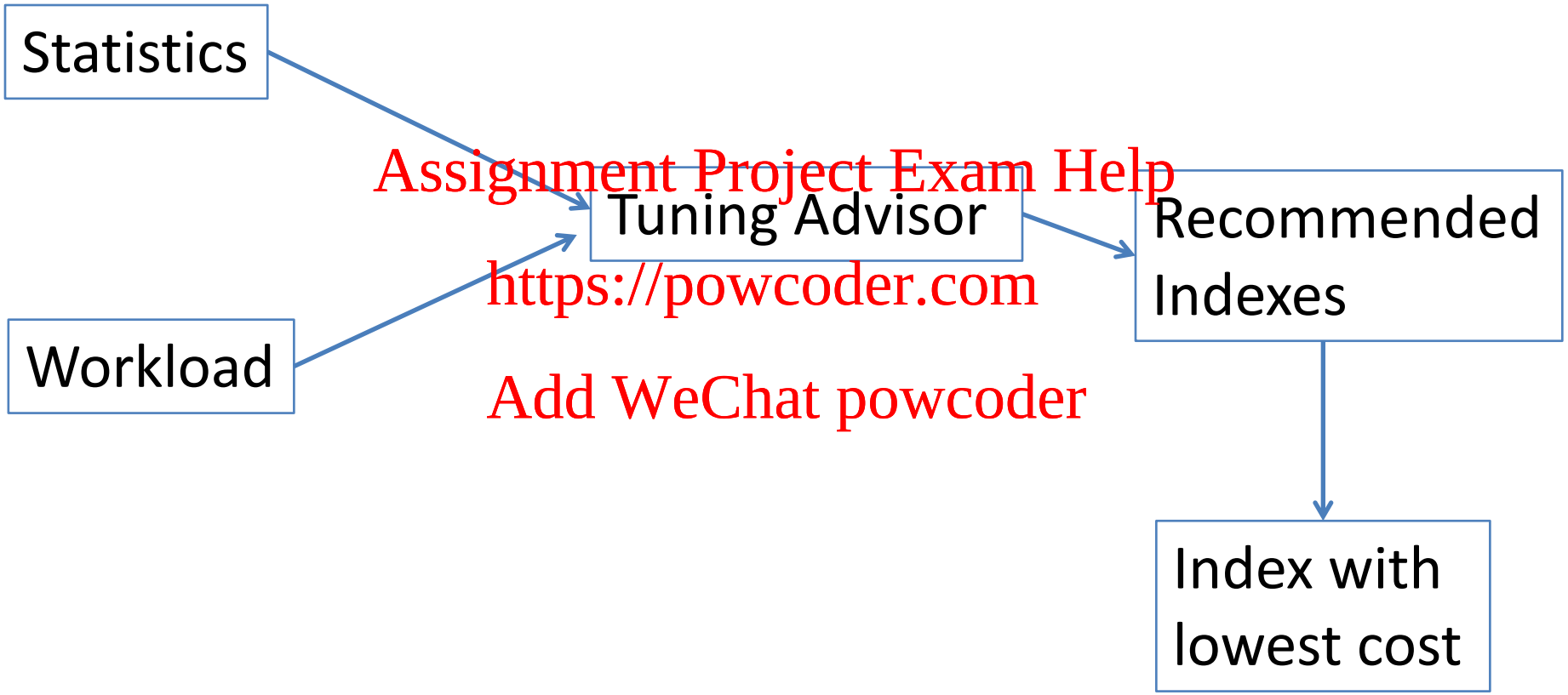
- With `CREATE INDEX YearIndex Movie(year)`
- This index facilitates the following search.


```
select * from Movies where year= 1990;
```


- What if Movies are not stored by year ?

  There will be little gain from the index on year. Imagine a scenario where movie tuples made on 1990 are spread all over the places on the disk.

Index Creation



Example: Index Creation

`StarIn(movieTitle,movieYear,starName)`

Assumptions:

- StarIn occupies 10 disk pages
- A star has appeared in 3 movies
- A movie has 3 stars
- One disk access is assumed to read a page of the index every time we use that index.
- Modification on the index needs two disk accesses (one to read the page and another to write it back to disk)

Example: Index Creation

```
Q1:
SELECT movieTitle, movieYear
FROM StarsIn
WHERE starName = 'Harrison Ford';
```


Index on starName Index on (title, year)

No Index	Star Index	Movie Index	Both Index
10	4 (1 index access + 3 tuple accesses)	10	4 (movie index doesn't help)

Example: Index Creation

```
Q2:
SELECT starName
FROM StarsIn
WHERE movieTitle=t AND movieYear = y;
```


Index on starName Index on (title, year)

No Index	Star Index	Movie Index	Both Index
10	10	4	4 (star index doesn't help)

Example: Index Creation

```
I:

INSERT INTO StarIn VALUES(t,y,s);
```


Index on starName Index on (title, year)

No Index	Star Index	Movie Index	Both Index
2	4 (2 for modifying data and 2 for modifying index)	4	6

Example: Index Creation

	No Index	Star Index	Movie Index	Both Index
Average number of disk accesses	$8P_1+8P_2+2$	$6P_2+4$	$6P_1+4$	$6-2P_1-2P_2$

P_1: Fraction of time doing Q1
P_2: Fraction of time doing Q2
I: $1 - P_1 - P_2$

Materialized Views

- Stored in database
- Benefits of using Virtual Views + faster query performance.
- In principle, we need to re-compute a materialized view every time one of its base tables changes
- Make changes to the materialized view incremental.

Incremental Changes to Materialized Views

```
CREATE MATERIALIZED VIEW MovieProd AS
    SELECT title, year, name
    FROM Movies, MovieExec
    WHERE producerC# = cert#;
```


Movies(title, year, producerC#)

MovieExec(name, cert#)

Example: Incremental Changes to Materialized Views

INSERT INTO Movies VALUES ('Kill Bill', 2003, 23456);

→

SELECT name

FROM MovieExec

where cert# = 23456; // only one name, say Smith, returns because cert# is the key

INSERT INTO MovieProd VALUES ('Kill Bill', 2003, 'Smith');

Example: Incremental Changes to Materialized Views

DELETE FROM Movies WHERE title = 'Bambi' and year = 1994;

→

DELETE FROM MovieProd

WHERE title = 'Bambi' and year = 1994;

Example: Incremental Changes to Materialized Views

INSERT INTO MovieExec VALUES ('Max', 34567);

→

INSERT INTO MovieProd

SELECT title, year, 'Max'

FROM Movies

WHERE prodecerC# = 34567;

Example: Incremental Changes to Materialized Views

DELETE FROM MovieExec WHERE cert#=45678;

→

DELETE FROM MovieProd

WHERE (title, year) IN

(SELECT title, year FROM Movies

WHERE producerC# = 45678);

Periodic Maintenance of Materialized Views