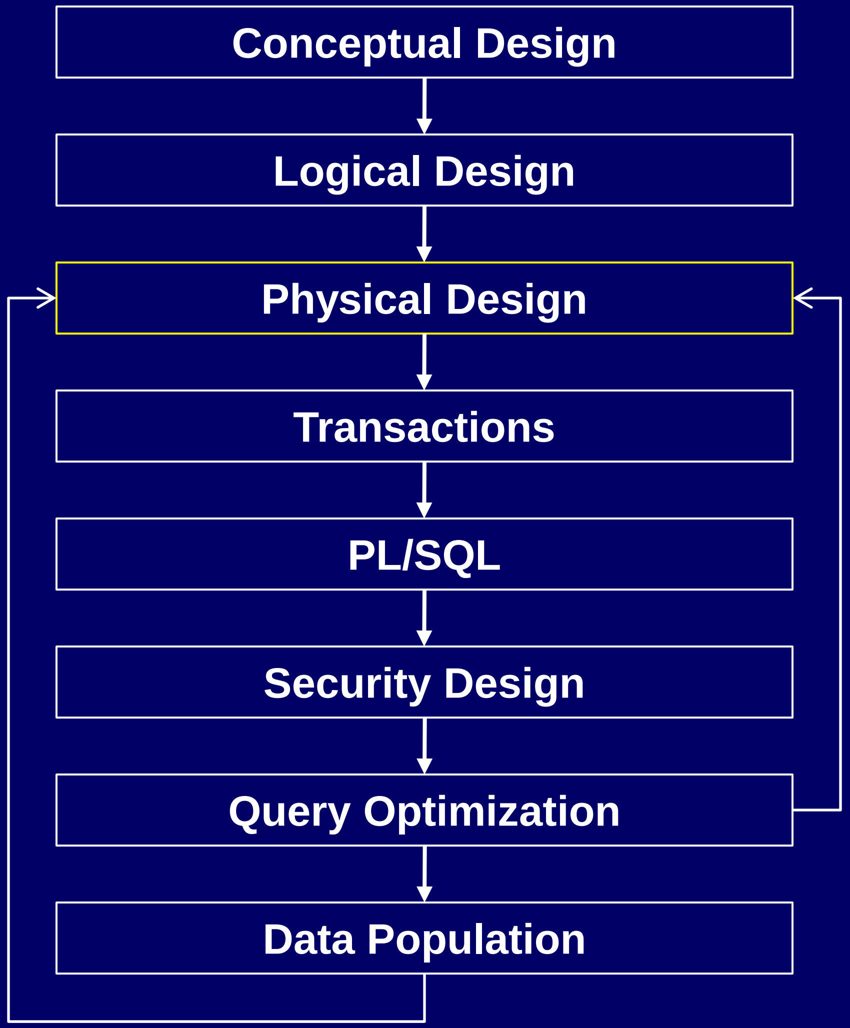
CSCD43: Database Systems Technology

Lecture 4

Wael Aboulsaadat

Acknowledgment: these slides are based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

Steps in Database Design



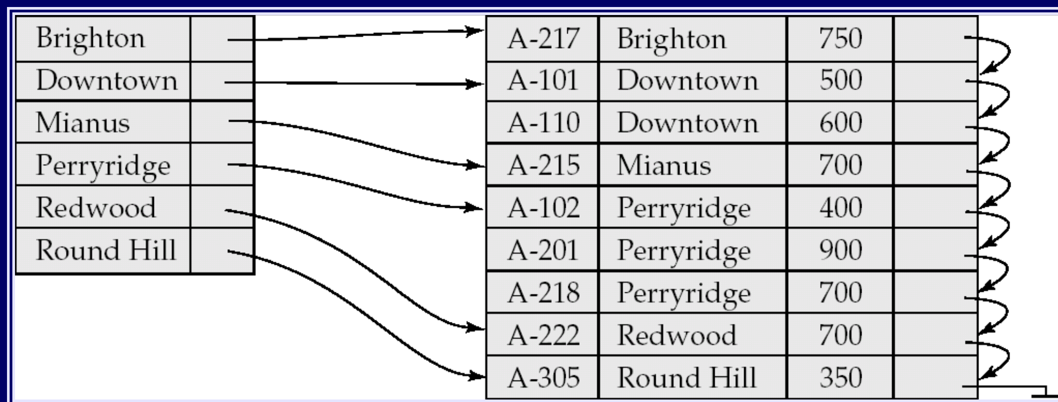
Physical Design

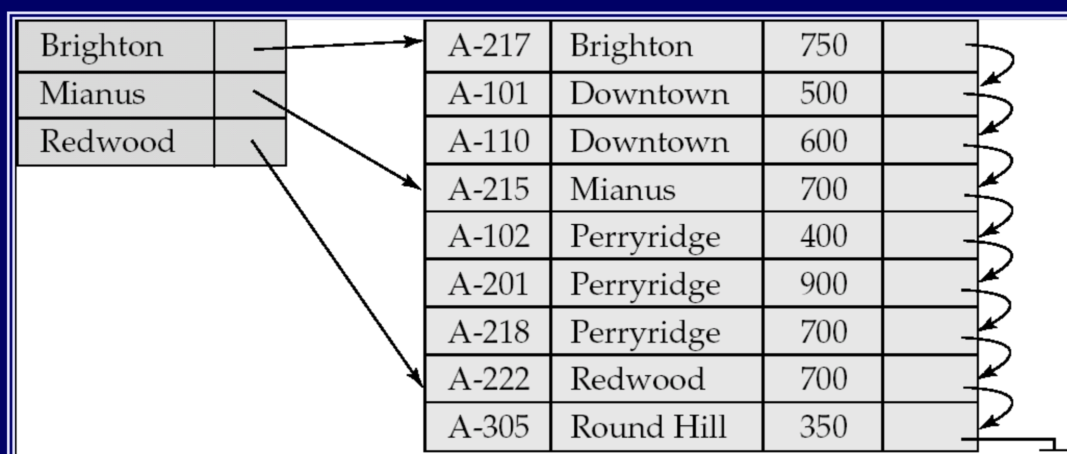
A. Specify Storage parameters

B. Specify Indices

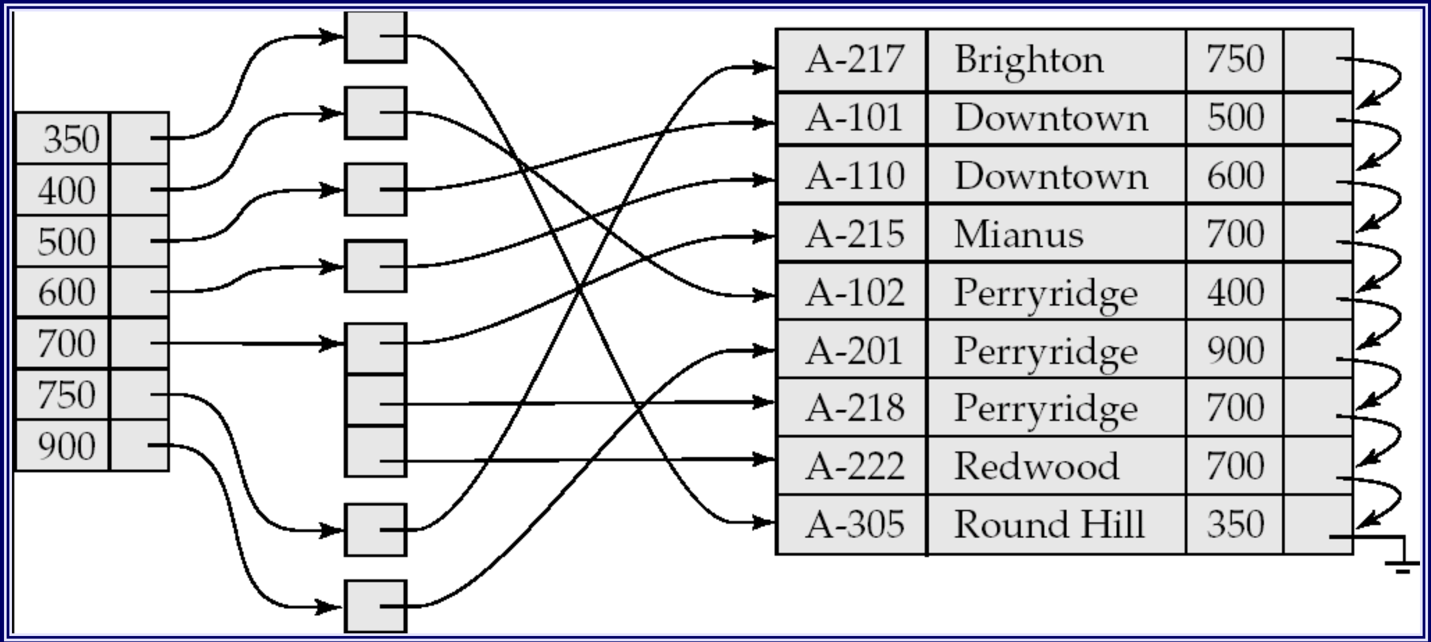
Storage and Indexing

Recall Index strategies





Recall Index strategies

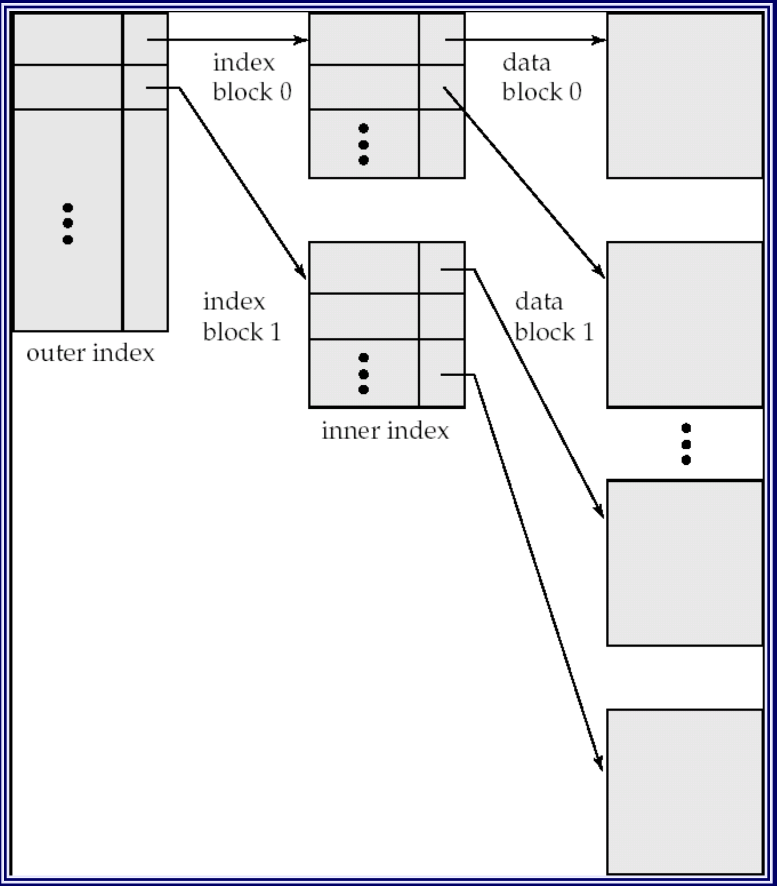


Question?

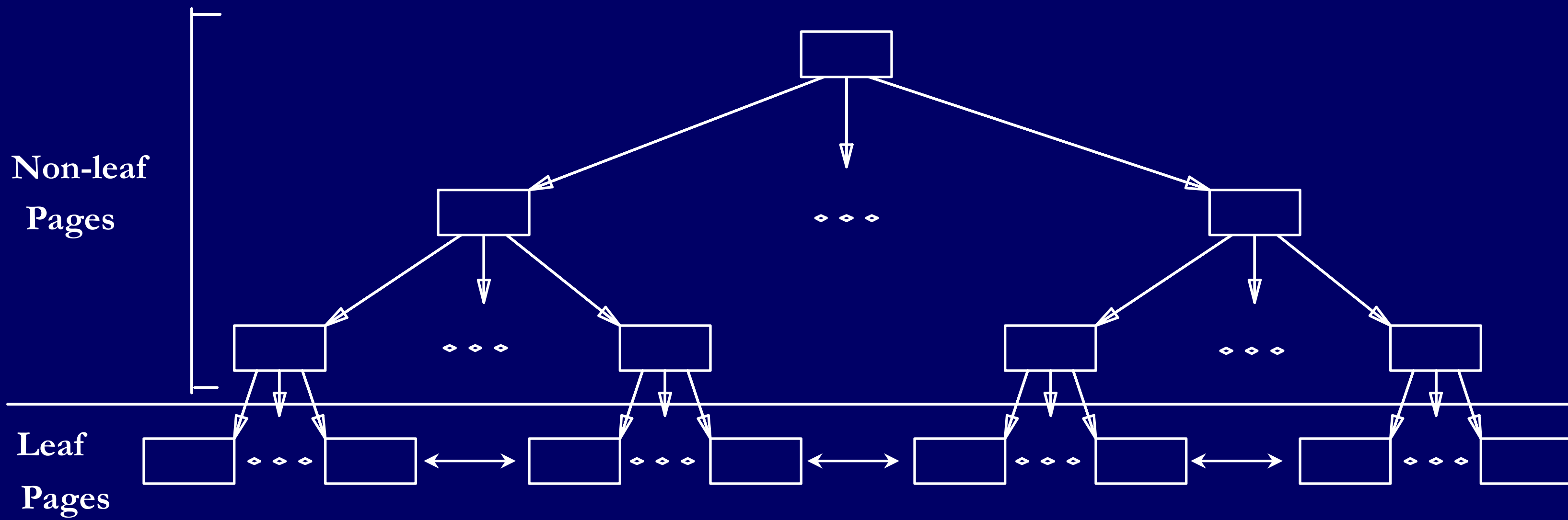
- What if index is too large to search sequentially?

Multi-level Index

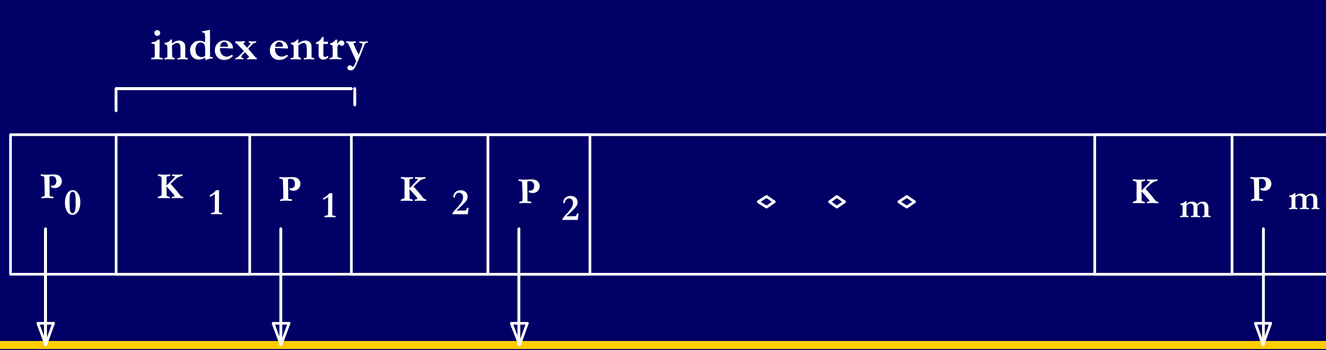
- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution?
  - Build an index on the index itself



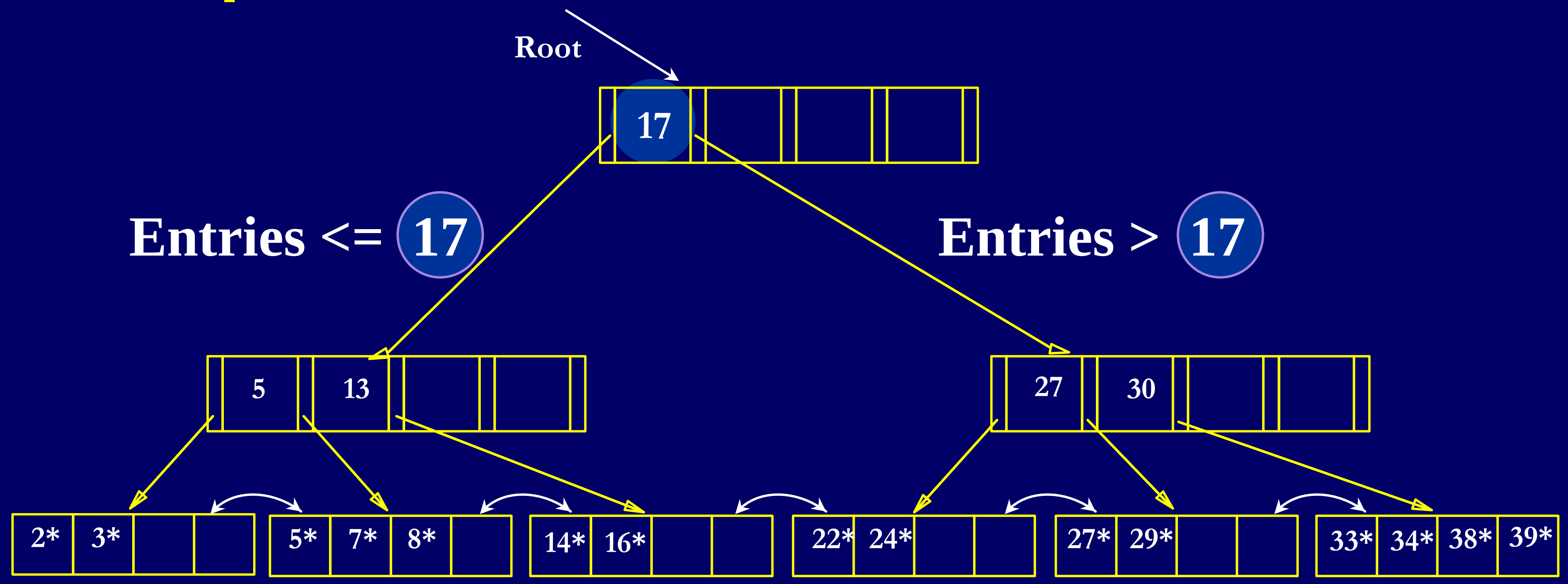
B+ Tree Indexes



- **Leaf** pages contain *data entries*, and are **chained** (prev & next)
- **Non-leaf** pages contain *index entries* and direct searches:

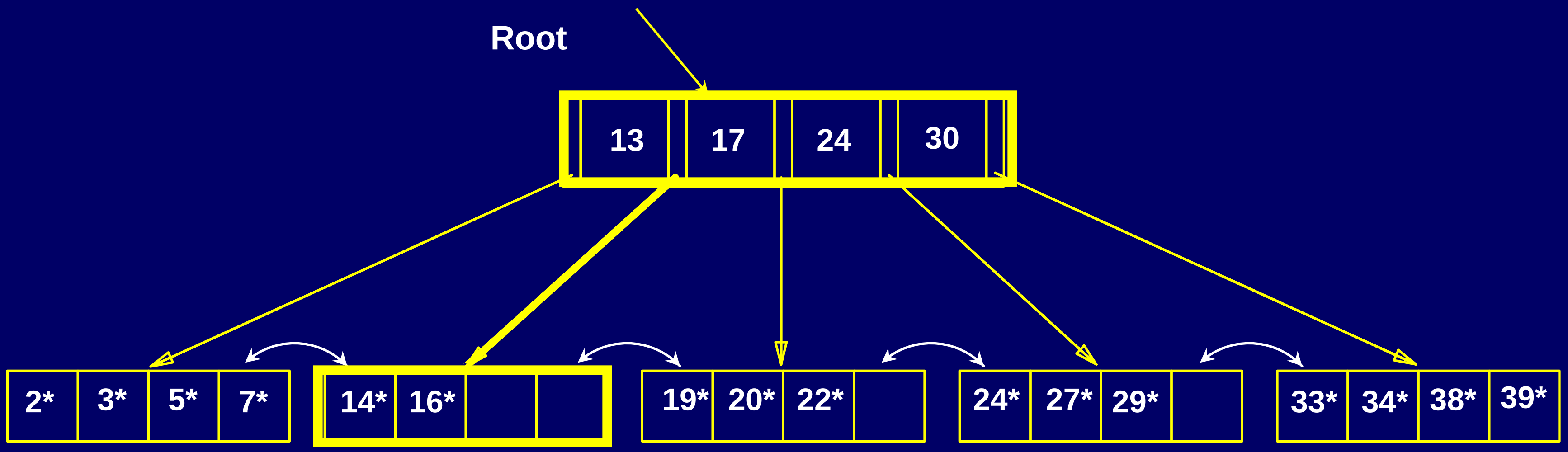


Example B+ Tree



B+ Tree Equality Search

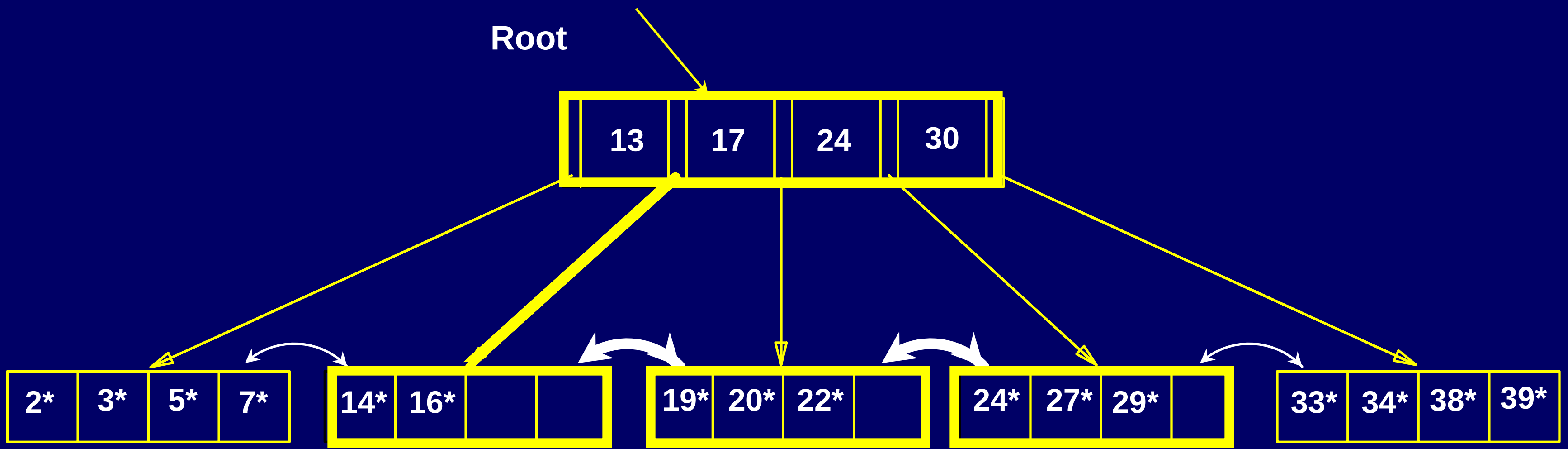
- Search begins at root, and key comparisons direct it to a leaf.
- Search for 15*…



✉ *Based on the search for 15*, we <u>know</u> it is not in the tree!*

B+ Tree Range Search

- Search all records whose ages are in [15,28].
  - Equality search 15*.
  - Follow sibling pointers.



How to create an index in SQL ?

- Syntax

  CREATE INDEX Index-Name on Table-Name(Columns…);

- Example:

  TABLE *Customer* (First_Name char(50),
  Last_Name char(50),
  Address char(50),
  City char(50),
  Country char(25),
  Birth_Date date)

  CREATE INDEX IDX_CUSTOMER_LAST_NAME on CUSTOMER (Last_Name)

  CREATE INDEX IDX_CUSTOMER_LOCATION on CUSTOMER (City, Country)

How to drop an index in SQL ?

- Syntax

```
DROP INDEX  Index-Name;
```

- Example:

```
DROP INDEX  IDX_CUSTOMER_LAST_NAME;

DROP INDEX IDX_CUSTOMER_LOCATION;
```

- PostgreSQL tables

```
select * from pg_index;
```

When to use an Index?

- Table contains a large number of records (a rule of thumb is that a large table contains over 100,000 records/tuples)
- The field contains a wide range of values
- The field contains a large number of NULL values
- Application queries frequently use the field in a search condition or join condition
- Most queries retrieve less than 5% of the table rows

- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

When not to use an Index?

- The table does not contain a large number of records
- Applications do not use the proposed index field in a query search condition
- Most queries retrieve more than 5% of the table records
- Applications frequently insert or modify table data

Physical Design

A. Specify Storage parameters
   - http://www.postgresql.org/docs/8.1/static/runtime-config.html
   - *Will discuss in few weeks*

A. Specify Indices

Transactions

The Setting

- Database systems are normally being accessed by many users or processes at the same time.
  - Both queries and modifications.
- Unlike Operating Systems, which *support* interaction of processes, a DMBS needs to keep processes from troublesome interactions.

Example: Bad Interaction

- You and your spouse each take $100 from different ATM’s at about the same time.
  - The DBMS better make sure one account deduction doesn’t get lost.
- Compare: An OS allows two people to edit a document at the same time. If both write, one’s changes get lost.

ACID Transactions

- A DBMS is expected to support "ACID transactions," which are:
  - *Atomic* : Either the whole process is done or none is.
  - *Consistent* : Database constraints are preserved.
  - *Isolated* : It appears to the user as if only one process executes at a time.
  - *Durable* : Effects of a process do not get lost if the system crashes.

Transactions in SQL

- SQL supports transactions, often behind the scenes.
  - Each statement issued at the generic query interface is a transaction by itself.
  - In programming interfaces like Embedded SQL or PSM, a transaction begins the first time an SQL statement is executed and ends with the program or an explicit end.

COMMIT

- The SQL statement COMMIT causes a transaction to complete.
  - It's database modifications are now permanent in the database.

ROLLBACK

- The SQL statement ROLLBACK also causes the transaction to end, but by *aborting*.
  - No effects on the database.
- Failures like division by 0 can also cause rollback, even if the programmer does not request it.

An Example: Interacting Processes

- Assume the usual Sells(bar,beer,price) relation, and suppose that Joe's Bar sells only Bud for $2.50 and Miller for $3.00.
- Sally is querying Sells for the highest and lowest price Joe charges.
- Joe decides to stop selling Bud and Miller, but to sell only Heineken at $3.50.

Sally's Program

- Sally executes the following two SQL statements, which we call (min) and (max), to help remember what they do.

```
(max)SELECT MAX(price) FROM Sells
            WHERE bar = 'Joe''s Bar';
(min) SELECT MIN(price) FROM Sells
            WHERE bar = 'Joe''s Bar';
```

Joe’s Program

- At about the same time, Joe executes the following steps, which have the mnemonic names (del) and (ins).

```
(del)     DELETE FROM Sells
          WHERE bar = 'Joe''s Bar';
(ins)     INSERT INTO Sells
          VALUES('Joe''s Bar', 'Heineken',
              3.50);
```

Interleaving of Statements

- Although (max) must come before (min) and (del) must come before (ins), there are no other constraints on the order of these statements, unless we group Sally's and/or Joe's statements into transactions.

Example: Strange Interleaving

- Suppose the steps execute in the order (max)(del)(ins)(min).

Joe's Prices:	2.50, 3.00	2.50, 3.00		3.50
Statement:	(max)	(del)	(ins)	(min)
Result:	3.00			3.50

- Sally sees MAX < MIN!

Fixing the Problem With Transactions

- If we group Sally's statements (max)(min) into one transaction, then she cannot see this inconsistency.
- She see's Joe's prices at some fixed time.
  - Either before or after he changes prices, or in the middle, but the MAX and MIN are computed from the same prices.

Another Problem: Rollback

- Suppose Joe executes (del)(ins), but after executing these statements, thinks better of it and issues a ROLLBACK statement.
- If Sally executes her transaction after (ins) but before the rollback, she sees a value, 3.50, that never existed in the database.

Solution

- If Joe executes (del)(ins) as a transaction, its effect cannot be seen by others until the transaction executes COMMIT.
  - If the transaction executes ROLLBACK instead, then its effects can *never* be seen.

Isolation Levels

- SQL defines four *isolation levels* = choices about what interactions are allowed by transactions that execute at about the same time.
- How a DBMS implements these isolation levels is highly complex, and a typical DBMS provides its own options.

Choosing the Isolation Level

- Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL *X*

where *X* =

1. SERIALIZABLE
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

Serializable Transactions

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level SERIALIZABLE, then she will see the database either before or after Joe runs, but not in the middle.
- It's up to the DBMS vendor to figure out how to do that, e.g.:
  - True isolation in time.
  - Keep Joe's old prices around to answer Sally's queries.

Isolation Level Is Personal Choice

- Your choice, e.g., run serializable, affects only how *you* see the database, not how others see it.
- Example: If Joe Runs serializable, but Sally doesn't, then Sally might see no prices for Joe's Bar.
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction.

Read-Commited Transactions

- If Sally runs with isolation level READ COMMITTED, then she can see only committed data, but not necessarily the same data each time.
- Example: Under READ COMMITTED, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits.
  - Sally sees MAX < MIN.

Repeatable-Read Transactions

- Requirement is like read-committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.

Example: Repeatable Read

- Suppose Sally runs under REPEATABLE READ, and the order of execution is (max)(del)(ins)(min).
  - (max) sees prices 2.50 and 3.00.
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max).

Read Uncommitted

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).
- Example: If Sally runs under READ UNCOMMITTED, she could see a price 3.50 even if Joe later aborts.

Transaction in PostgreSQL

- Syntax

```
BEGIN;
        …….
COMMIT;
```

- Example

```
BEGIN;
        UPDATE accounts SET balance = balance - 100.00
              WHERE name = 'Alice';
        -- etc etc….
COMMIT;
```

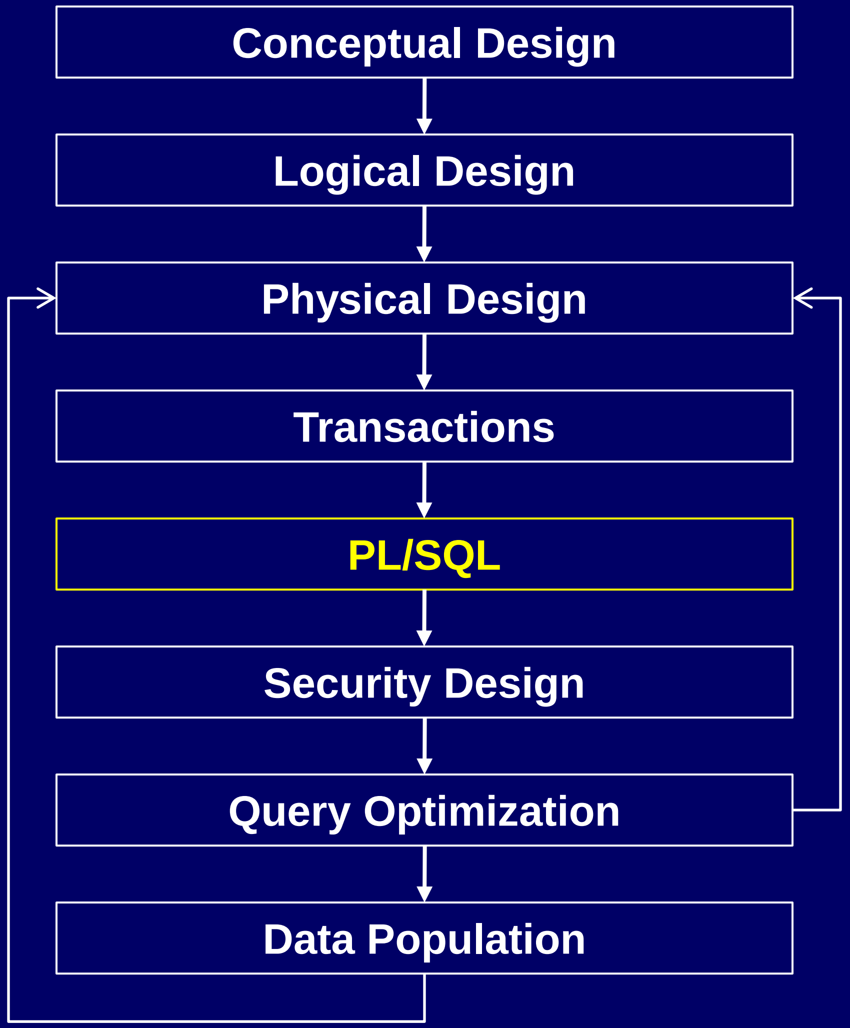
Transaction in PostgreSQL

- Example

```
BEGIN;
    UPDATE accounts SET balance = balance - 100.00 WHERE name
                         = 'Alice';
    SAVEPOINT my_savepoint;
    UPDATE accounts SET balance = balance + 100.00 WHERE
                         name = 'Bob';
    -- oops ... forget that and use Wally's account
    ROLLBACK TO my_savepoint;
    UPDATE accounts SET balance = balance + 100.00 WHERE
              name = 'Wally';
COMMIT;
```

SQL/PSM/PL-SQL

Steps in Database Design

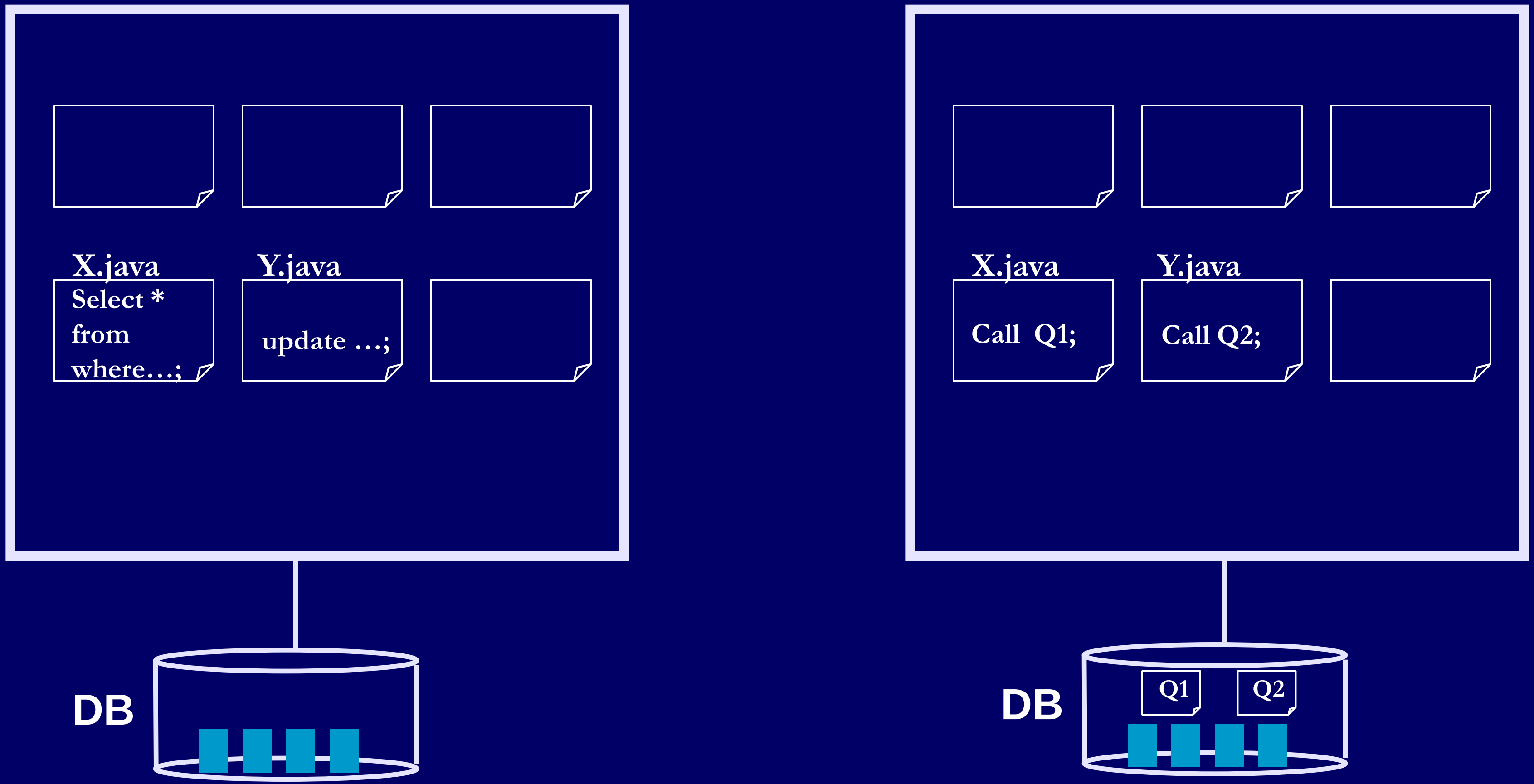


Stored Procedures

- An extension to SQL, called SQL/PSM, or “persistent, stored modules,” allows us to store procedures as database schema elements.
- The programming style is a mixture of conventional statements (if, while, etc.) and SQL.
- Let’s us do things we cannot do in SQL alone.

Stored Procedures

- A great technique for enhancing modularity of software



Basic PSM Form

```
CREATE PROCEDURE <name> (
        <parameter list> )
   <optional local declarations>
   <body>;
```

- Function alternative:

```
CREATE FUNCTION <name> (
        <parameter list> ) RETURNS <type>
```

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = procedure uses value, does not change value.
  - OUT = procedure changes, does not use.
  - INOUT = both.

Example: Stored Procedure

- Let's write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells that has bar = 'Joe''s Bar', beer = *b*, and price = *p*.
  - Used by Joe to add to his menu more easily.

The Procedure

```
CREATE PROCEDURE JoeMenu (
    IN b      CHAR(20),
    IN p      REAL
)
INSERT INTO Sells
VALUES('Joe''s Bar', b, p);
```

Parameters are both read-only, not changed

The body --- a single insertion

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments.
- Example:

  CALL JoeMenu('Moosedrool', 5.00);

- Functions used in SQL expressions where a value of their return type is appropriate.

Types of PSM statements -- 1

- RETURN <expression> sets the return value of a function.
  - Unlike C, etc., RETURN *does not* terminate function execution.
- DECLARE <name> <type> used to declare local variables.
- BEGIN . . . END for groups of statements.
  - Separate by semicolons.

Types of PSM Statements -- 2

- Assignment statements:

  SET <variable> = <expression>;

  - Example: SET b = ’Bud’;
- Statement labels: give a statement a label by prefixing a name and a colon.

IF statements

- Simplest form:

    IF <condition> THEN

        <statements(s)>

    END IF;

- Add ELSE <statement(s)> if desired, as

    IF . . . THEN . . . ELSE . . . END IF;

- Add additional cases by ELSEIF <statements(s)>:

IF … THEN … ELSEIF … ELSEIF … ELSE … END IF;

Example: IF

- Let's rate bars by how many customers they have, based on Frequents(drinker, bar).
  - <100 customers: 'unpopular'.
  - 100-199 customers: 'average'.
  - >= 200 customers: 'popular'.
- Function Rate(b) rates bar b.

Example: IF (continued)

```
CREATE FUNCTION Rate (IN b CHAR(20) )
      RETURNS CHAR(10)
      DECLARE cust INTEGER;
   BEGIN
      SET cust = (SELECT COUNT(*) FROM Frequents
                    WHERE bar = b);
      IF cust < 100 THEN RETURN ’unpopular’
      ELSEIF cust < 200 THEN RETURN ’average’
      ELSE RETURN ’popular’
      END IF;
   END;
```

Number of customers of bar b

Nested IF statement

Return occurs here

Loops

- Basic form:

  LOOP <statements> END LOOP;

- Exit from a loop by:

  LEAVE <loop name>

- The <loop name> is associated with a loop by prepending the name and a colon to the keyword LOOP.

Example: Exiting a Loop

```
loop1: LOOP
        . . .
        LEAVE loop1;      ← If this statement is executed . . .
        . . .
END LOOP;                 ← Control winds up here
```

Other Loop Forms

- WHILE <condition>
      DO <statements>
  END WHILE;
- REPEAT <statements>
  UNTIL <condition>
  END REPEAT;

Queries

- General SELECT-FROM-WHERE queries are *not* permitted in PSM.
- There are three ways to get the effect of a query:
  1. Queries producing one value can be the expression in an assignment.
  2. Single-row SELECT . . . INTO.
  3. Cursors.

Example: Assignment/Query

- If *p* is a local variable and Sells(bar, beer, price) the usual relation, we can get the price Joe charges for Bud by:

```
SET p = (SELECT price FROM Sells
   WHERE bar = ’Joe’’s Bar’ AND
         beer = ’Bud’);
```

SELECT . . . INTO

- An equivalent way to get the value of a query that is guaranteed to return a single tuple is by placing INTO <variable> after the SELECT clause.
- Example:

```
SELECT price INTO p FROM Sells
WHERE bar = ’Joe’’s Bar’ AND
    beer = ’Bud’;
```

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Declare a cursor *c* by:

DECLARE c CURSOR FOR <query>;

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

  OPEN c;

  - The query of *c* is evaluated, and *c* is set to point to the first tuple of the result.
- When finished with *c*, issue command:

  CLOSE c;

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

  FETCH FROM c INTO x1, x2,…,x*n* ;
- The *x* ’s are a list of variables, one for each component of the tuples referred to by *c*.
- c is moved automatically to the next tuple.

Breaking Cursor Loops -- 1

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

Breaking Cursor Loops -- 2

- Each SQL operation returns a *status*, which is a 5-digit number.
  - For example, 00000 = “Everything OK,” and 02000 = “Failed to find a tuple.”
- In PSM, we can get the value of the status in a variable called SQLSTATE.

Breaking Cursor Loops -- 3

- We may declare a condition, which is a boolean variable that is true if and only if SQLSTATE has a particular value.
- Example: We can declare condition NotFound to represent 02000 by:

```
DECLARE NotFound CONDITION FOR
   SQLSTATE '02000';
```

Breaking Cursor Loops -- 4

- The structure of a cursor loop is thus:

```
cursorLoop: LOOP
    …
    FETCH c INTO … ;
    IF NotFound THEN LEAVE cursorLoop;
    END IF;
    …
END LOOP;
```

Example: Cursor

- Let’s write a procedure that examines Sells(bar, beer, price), and raises by $1 the price of all beers at Joe’s Bar that are under $3.
  - Yes, we could write this as a simple UPDATE, but the details are instructive anyway.

The Needed Declarations

```
CREATE PROCEDURE JoeGouge( )
    DECLARE theBeer CHAR(20);
    DECLARE thePrice REAL;
    DECLARE NotFound CONDITION FOR
        SQLSTATE '02000';
    DECLARE c CURSOR FOR
        (SELECT beer, price FROM Sells
         WHERE bar = 'Joe''s Bar');
```

Used to hold beer-price pairs when fetching through cursor c

Returns Joe's menu

The Procedure Body

```
BEGIN
    OPEN c;
    menuLoop: LOOP
        FETCH c INTO theBeer, thePrice;
        IF NotFound THEN LEAVE menuLoop END IF;
        IF thePrice < 3.00 THEN
            UPDATE Sells SET price = thePrice+1.00
            WHERE bar = 'Joe''s Bar' AND beer = theBeer;
        END IF;
    END LOOP;
    CLOSE c;
END;
```

Check if the recent FETCH failed to get a tuple

If Joe charges less than $3 for the beer, raise it's price at Joe's Bar by $1.