PA152: Efficient Use of DB

12. Advanced Topics

sequences, spatial indexes, access control

Vlastislav Dohnal

Credits

- Materials are based on presentations:
  - Courses CS245, CS345, CS345
    - Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom
    - Stanford University, California
  - Course CS145 following the book
    - Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: Database Systems: The Complete Book
  - Book
    - Andrew J. Brust, Stephen Forte:
      Mistrovství v programování SQL Serveru 2005
  - MSDN library by Microsoft

Contents

- Generating IDs
- Spatial data
  - Data types, indexing
- DB security
  - Access control in DB
  - Stored procedures
  - Attacking DBMS

Generating PK values

- Typically, a sequence of numbers
  - Increasing monotonically
- Example:
  - student(učo, first_name, last_name)
- Ad-hoc solution 1:
    - Getting current maximum
      maxučo := SELECT max(učo) FROM student;
    - Incrementing and using in new record
      INSERT INTO student
      VALUES (maxučo+1, ‘Mad’, ‘Max’);
  - Disadvantage:
    - Concurrent use → duplicate values

Generating PK values

- Ad-hoc solution 2:
  - Combining INSERT and SELECT in a statement

    ```
    INSERT INTO student VALUES (
            (SELECT max(učo) FROM student)+1,
            ‘Mad’, ‘Max’);
    ```
  - Updates to index are atomic
    - Looks promising….
    - Nested select may be evaluated on “stale data”
  - Duplicate values are less probable.
    - Improved performance only
      - i.e., sending one statement to DB

Generating PK values

- Solution 2: issues in concurrency
  - Always when in transaction
  - Depends on way of locking DB uses:
    - SELECT locks data (shared lock)
      - Others are blocked
      - Locks are always released after commit
    - INSERT
    - → values are correct (no dups), but others are waiting

Generating PK values

- Ad-hoc solution 3:
  - □ Auxiliary table
    keys(table VARCHAR, id INTEGER)

1. UPDATE keys SET id=id+1
   WHERE table=‘student’;
2. newid := SELECT id FROM keys
                    WHERE table=‘student’;
   - □ Or one statements:
     newid := UPDATE keys SET id=id+1
                    WHERE table=‘student’ RETURNING id;
3. INSERT INTO student
   VALUES (newid , ‘Mad’, ‘Max’);

Generating PK values

- Solution 3:
  - Inconvenience in concurrency when in transaction:
    - UPDATE locks the record in *keys*
    - Locks get released after commit (after INSERT)
    - → values are correct (no dups), but others are waiting
  - Advantage:
    - If combined with Solution 1
      - i.e., two consecutive transactions
    - → values are correct (no dups) and nobody is blocked!

Generating PK values

- Recommended to use DB tools
  - Data types
    - PostgreSQL: SERIAL, BIGSERIAL
    - SQLServer: IDENTITY
  - Sequences
    - Oracle, PostgreSQL
  - Toggle at attribute
    - MySQL
- Support for getting last generated number
  - Good for inserting to tables with foreign keys
    - E.g., inserting first item into e-shopping basket
      - Creating a new basket & inserting goods

Generating PK values

- CREATE SEQUENCE …
  - Numeric sequence generator
  - Is parameterized:
    - Min / max value, cyclic
- Functions in PostgreSQL
  - Nextval – generate new value
  - Currval – get last generated value
  - Can be imbedded in INSERT
    - INSERT INTO table_name
      VALUES (nextval('sequence_name'), …);

Generating PK values: Performance

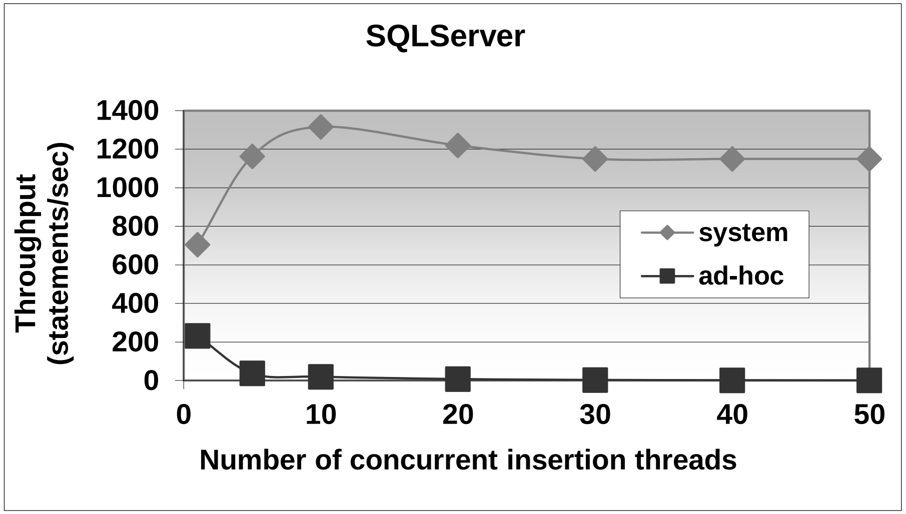
- Example for Solution 3:
  - □ accounts(number, branchnum, balance);
    - Clustered index on *number*
  - □ counter(nextkey);
    - One record with value 1
    - For generating values of *id* by Solution 3
- Configuration:
  - □ Transaction isolation: READ COMMITTED
    - Only committed data are visible.
  - □ Dual Xeon (550MHz,512Kb), 1GB RAM, RAID controller, 4x 18GB drives (10000RPM), Windows 2000.

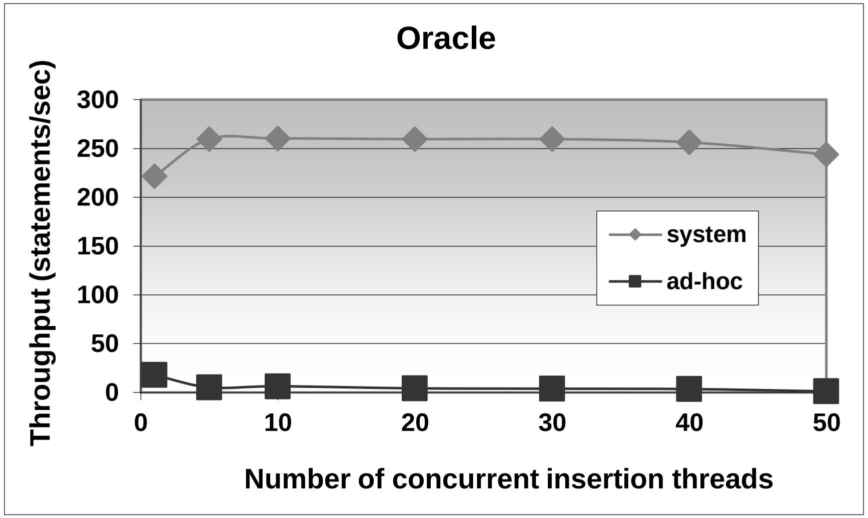
Generating PK values: Performance

- Batch of 100 000 insertions into *accounts*
- Generating ID values:
  - DB support:
    - SQLServer 7 (identity)
      - insert into accounts (branchnum, balance) values (94496, 2789);
    - Oracle 8i (sequence)
      - insert into accounts values (seq.nextval, 94496, 2789);
  - Solution 3:

```
begin transaction
   update counter  set nextkey = nextKey+1;
   :nk := select nextkey  from counter;
commit transaction
begin transaction
   insert into accounts values( :nk, 94496, 2789);
commit transaction
```

Generating PK values





- X axis:
  - Increasing number of parallel insertions
- *DB tools* outperforms *ad-hoc* solution.

Generating PK values

- PostgreSQL
  -
    ```
    CREATE TABLE product (
                    id SERIAL PRIMARY KEY,
                    title VARCHAR(10)
    );
    ```
  - Internal implementation
    - Create new sequence
      - product_id_seq
    - Attribute *id* has defaults value:
      - nextval('product_id_seq')

Generating PK values

- PostgreSQL (hand-crafted)

  -
    ```
    CREATE SEQUENCE product_id_seq;
    ```

  -
    ```
    CREATE TABLE product (
                id INT PRIMARY KEY
                        DEFAULT nextval(‘product_id_seq’),
                 title VARCHAR(10)
    );
    ```
- Usage:

  -
    ```
    INSERT INTO product (title)
                VALUES (‘Coil’);
    ```

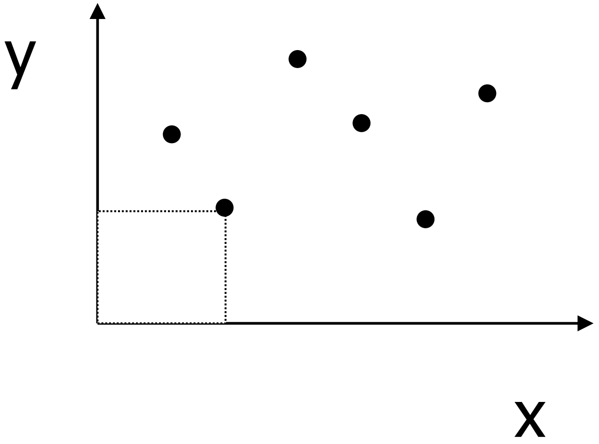
  -
    ```
    INSERT INTO product (id, title)
                VALUES (DEFAULT, ‘Coil’);
    ```

Contents

- Generating IDs
- **Spatial data**
  - **Data types, indexing**
- DB security
  - Access control in DB
  - Stored procedures
  - Attack on DB

Processing Spatial Data

- Spatial data
  - Typically geographic, 2d geometry
    - X, Y coordinates



<X_1,Y_1, Attributes>
<X_2,Y_2, Attributes>
…

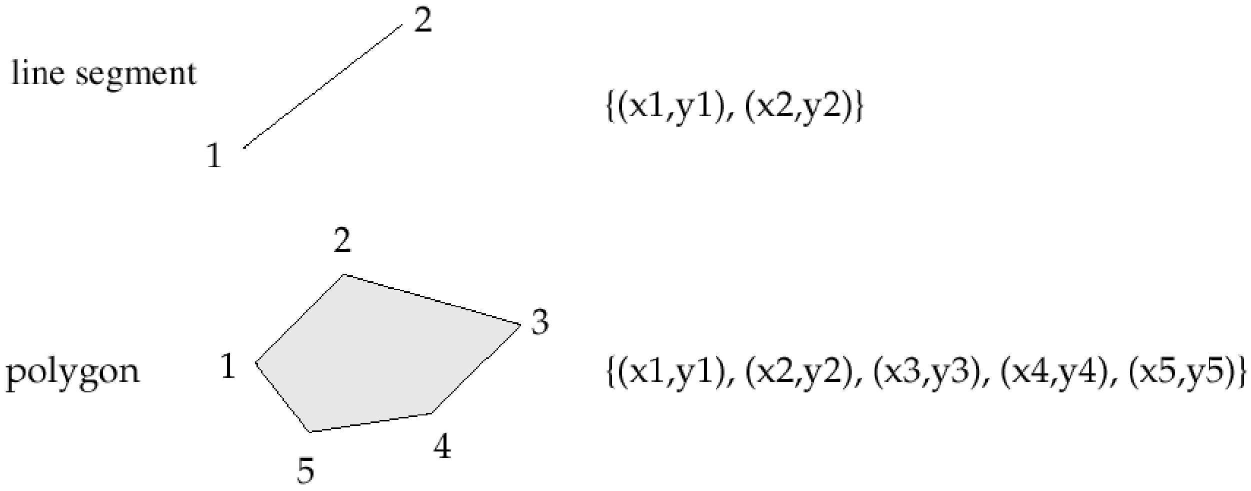
Processing Spatial Data

- Spatial queries
  - What city is at position <X_i,Y_i>?
  - What is in neighborhood of 5km from position <X_i,Y_i>?
  - What is the closest site to <X_i,Y_i>?

- Without DB support
  - How to measure distance? (e.g., for GPS coordinates)
    - Can create user-defined function
  - Index on X, or on XY, …
    - May not help for some queries

Processing Spatial Data

- Geometric constructs:
  - lines, rectangles, polygons, …



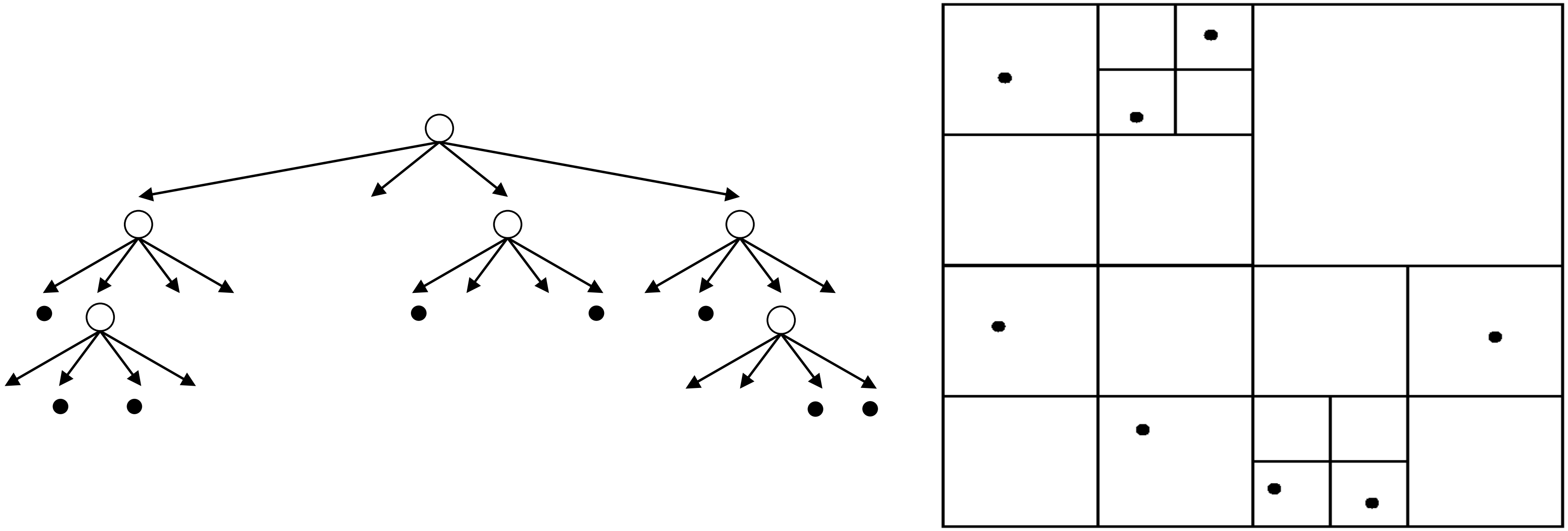
- Operations:
  - Is point inside a polygon? Do polygons intersect? …

Processing Spatial Data

- DB support is convenient
  - Special data types and functions/operators
    - PostgreSQL
      - Types: point, line, box, circle, …
      - Functions: area(), center(), length(), …
      - Operators: **~=** *same as*, **~** *contains*, **?#** *intersects*, …
      - Index: R-tree
    - SQL Server 2008
      - Types: point, linestring, polygon, geography, …
      - Index: Grid
    - Oracle 9i
      - Types: SDO_GEOMETRY (SDO_POINT, SDO_LINE,…)
      - Index: R-tree, Quad-tree

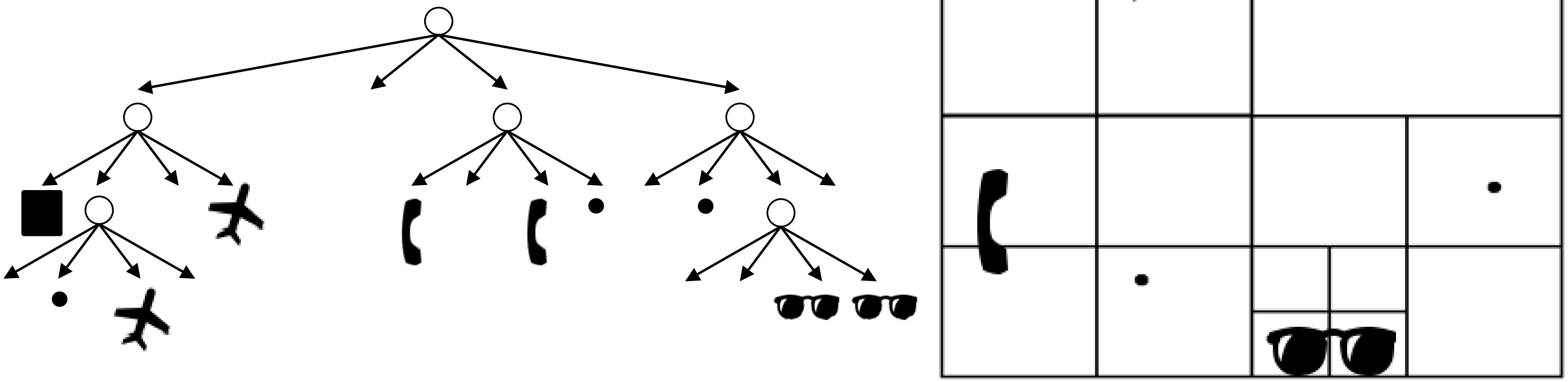
Processing Spatial Data

- Quad-tree
  - □ Search tree, where each node splits data space into 2^d regions of equal size
    - e.g., 2d data → 4 regions
  - □ Leaf nodes may be of larger capacity than 1.



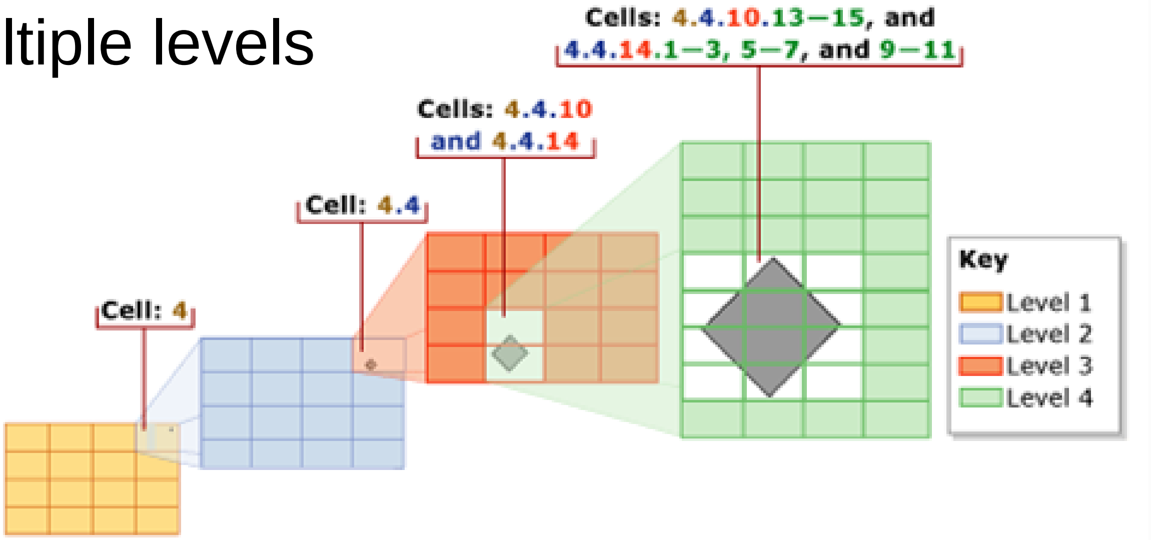
Processing Spatial Data

- Quad-tree
  - Supports points only
  - Extension to complex data:
    - Item stored in many regions
    - Complex objects wrapped in rectangle



Processing Spatial Data

- Grid
  - Bounded data space: x_{min}, y_{min}, x_{max}, y_{max}
  - SQL Server
    - Grid of fixed dimensions: 4x4, 8x8, 16x16 cells
    - Multiple levels



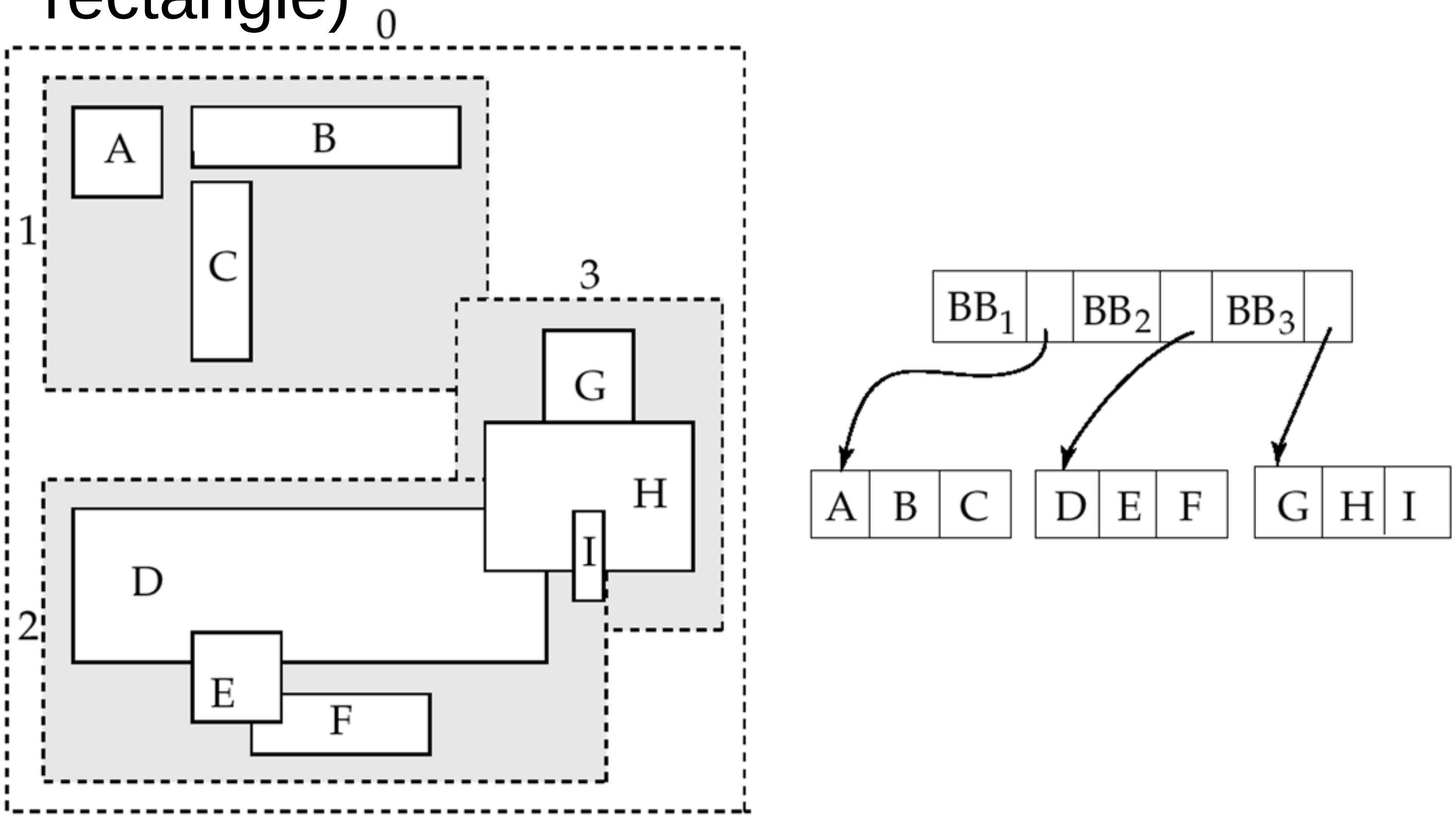
Zdroj: Microsoft MSDN, http://msdn.microsoft.com/en-us/library/bb964712.aspx

Processing Spatial Data

- R-tree (Rectangle Tree)
  - □ Extension of B^+ trees to *d*-dimensional data
    - Insertion, deletion – almost identical to B^+ tree
  - □ Leaves may contain more data items
    - List is represented by *minimum bounding rectangle (MBR)*
  - □ Internal nodes
    - References to child nodes and their MBRs
  - □ Node MBRs may overlap → search procedure has to follow more colliding tree branches.
  - □ Each data item stored exactly once
    - Advantage over Grid and Quad-tree

Processing Spatial Data

- R-tree
  - Organizing complex spatial data done by wrapping them in MBR (object represented as a rectangle)



Contents

- Generating IDs
- Spatial data
  - Data types, indexing
- **DB security**
  - **Access control in DB**
  - Stored procedures
  - Attack on DB

Access Control – Authorization

- Analogy to file systems
  - Objects
    - File, directory, …
  - Subject
    - Typically: owner, group, others (all users)
  - Access Right
    - Defined on a an object *O* for a subject *S*
    - Typically: read, write, execute

Privileges

- Database systems
  - Typically finer granularity than the typical file system
  - Varies for objects
    - Tables, views, sequences, schema, database, procedures, …
  - Views
    - an important tool for access control
  - Subjects are typically user and group
    - Often referred as *authorization id* or *role*
    - Subject “others“ is denoted as PUBLIC
      - Granting access for PUBLIC means allowing access to anyone.

Privileges

- For relations/tables:
  - SELECT
    - query the table’s content (i.e. print rows)
    - Sometimes can be limited to selects attributes
  - INSERT
    - Sometimes can be limited to selects attributes
  - DELETE
  - UPDATE
    - Sometimes can be limited to selects attributes
  - REFERENCES
    - creating foreign keys referencing this table

Privileges

- Example
  - INSERT INTO Beers(name)

    ```
    SELECT beer FROM Sells
    WHERE NOT EXISTS
            (SELECT * FROM Beers
             WHERE name = beer);
    ```

    We add beers that do not appear in Beers; leaving manufacturer NULL.
  - Requirements for privileges:
    - INSERT on the table *Beers*
    - SELECT on *Sells* and *Beers*

Privileges

- Views as Access Control
  - Relation
    - Employee(id, name, address, salary)
  - Want to make salary confidential:
    - CREATE VIEW EmpAddress AS

      ```
      SELECT id, name, address
      FROM Employee;
      ```
    - Privileges:
      - Revoke SELECT from table Employee
      - Grant SELECT on EmpAddress

Privileges

- Granting privileges
  - GRANT <list of privileges>
    ON <relation or object>
    TO <list of authorization ID’s>;
- You may also grant “grant privilege”
  - By appending clause “WITH GRANT OPTION“
    - GRANT SELECT
      ON TABLE EmpAddress
      TO karel
      WITH GRANT OPTION

Privileges

- Example (to be run as owner of *sells*)
  - GRANT SELECT, UPDATE(price)
    ON sells TO sally;
- User *sally* can
  - Read (select) from table *sells*
  - Update values in attribute *price*

Privileges

- Example (to be run as owner of *sells*)
  - GRANT UPDATE ON sells TO sally WITH GRANT OPTION;
- User *sally* can
  - Update values of any attribute in *sells*
  - Grant access to other users
    - Only UPDATE can be granted, but can be limited to some attributes.

Privileges

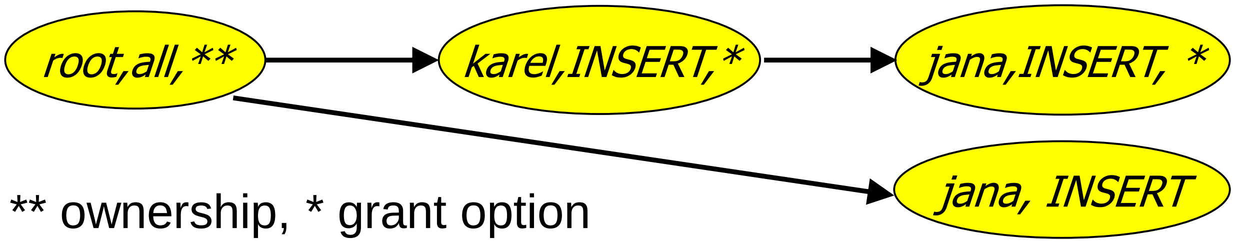
- Revoking statement
  - REVOKE <list of privileges>
    ON <relation or object>
    FROM <list of authorization ID’s>;
- Listed users can no longer use the priviledges.
  - But they may still have the privilege
  - → because they obtained it independently from elsewhere.
    - Or they are members of a group or PUBLIC is applied

Privileges

- Revoking privileges
  - Appending to REVOKE statement:
    - CASCADE – Now, any grants made by a revokee are also not in force, no matter how far the privilege was passed
    - RESTRICT (implicit) –
      - If the privilege has been passed to others, the REVOKE fails as a warning
      - So something else must be done to “chase the privilege down.”
  - REVOKE GRANT OPTION FOR …
    - Removes the “grant option” only.
    - Omitting this leads to removing the privilege and also the grant option!

Privileges – Diagram

- Diagram depict privileges granted by a grantor to a grantee



  - Each object has its diagram
  - Node is specified by
    - Role (user / group)
    - Granted privilege
    - Flag of ownership or granting option
  - Edge from X to Y
    - X has granted the privilege to Y

Privileges – Diagram

- „*root,all*“ denotes
  - user *root* has privilege *all*.
- Privilege „*all*“ on table means
  - = insert, update, delete, select, references
- Grant option “*“
  - The privilege can by granted by the user
- Option “**“
  - Object owner (root node of each diagram)
- Object owner
  - All is granted by default
  - Can pass the privileges to other users

Privileges – Diagram

- Manipulating edges
  - When *A* grants *P* to *B*, We draw an edge from *AP* * or *AP* ** to *BP*.
    - Or to *BP* * if the grant is with grant option.
  - If *A* grants a subprivilege *Q* of *P* then the edge goes to *BQ* or *BQ* *, instead.
    - Q can be “UPDATE(a) on R”,
      whereas *P* is “UPDATE ON R”

Privileges – Diagram

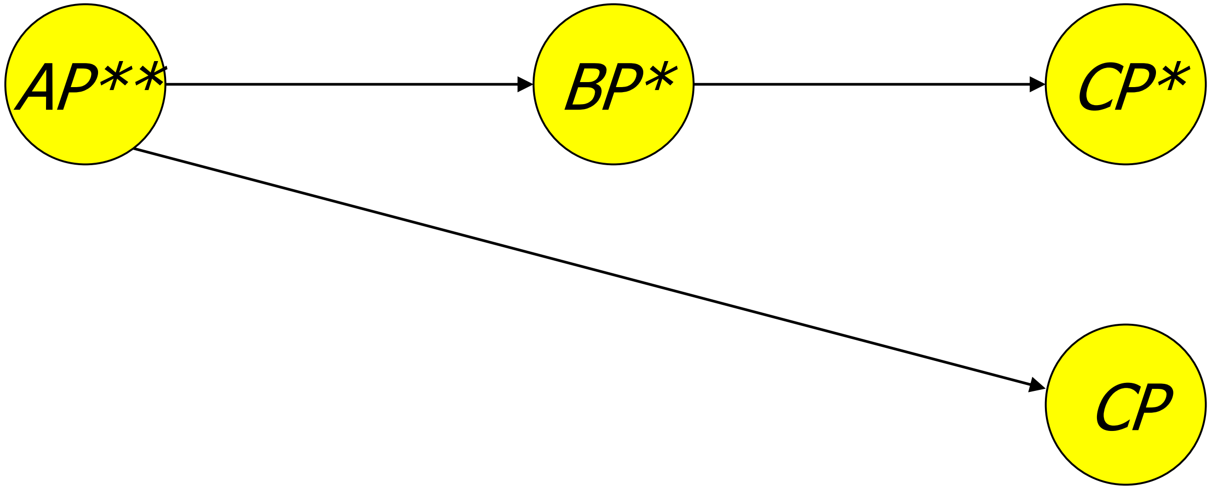
- Test for access
  - User *C* has privilege *Q* as long as there is a path from *XP*** to *OP*, *OP** nebo *OP***, where
    - *P* is superprivilege of *Q* or the same as *Q*, and
    - *O* = *C* or *C* is a member of group *O*

Privileges – Diagram

A owns the object
on which P is a
privilege.

A:
GRANT P TO B
WITH GRANT OPTION

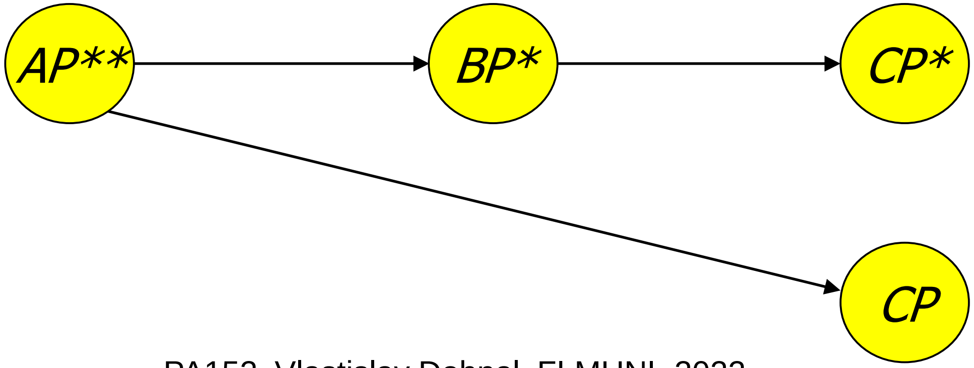
B:
GRANT P TO C
WITH GRANT OPTION



A:
GRANT P TO C

Privileges – Diagram

- Revoking privileges
  - If *A* revokes *P* from *B*
    - Test whether there is an edge AP → BP.
    - If so, edge is deleted.
  - If B granted P to someone else, CASCADE must be appended.



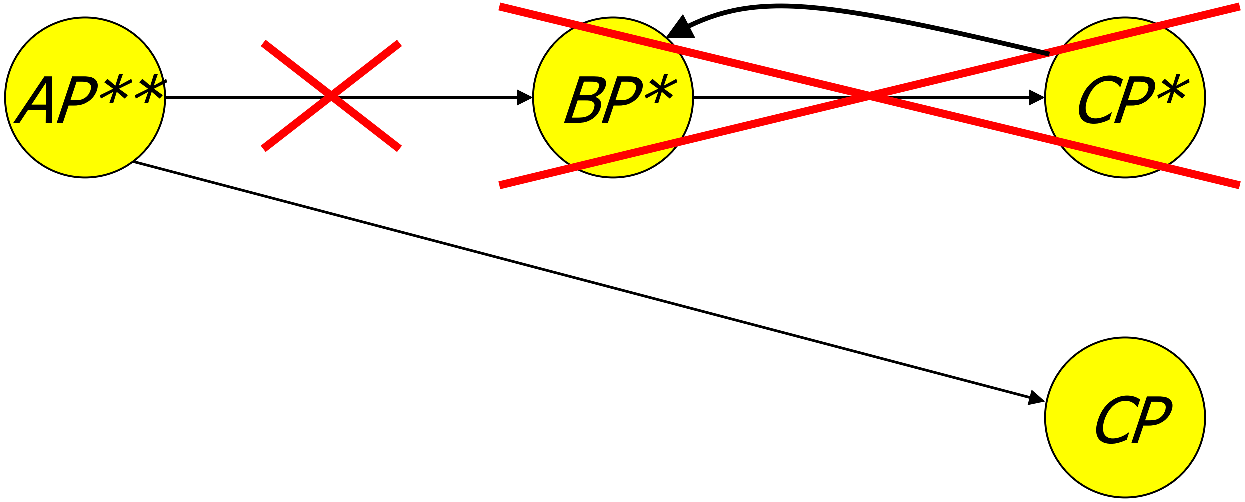
Privileges – Diagram

- Revoking privileges
  - Having deleted an edge, we must check
    - each node has a path from the ** node, representing ownership.
  - Any node with no such path represents a revoked privilege
    - So it is deleted from the diagram including all edges from it.

Privileges – Diagram

A:
REVOKE P FROM B
CASCADE

Not only does *B* lose *P**, but *C* loses *P**. Delete nodes *BP** and *CP**.



Even had *C* passed *P** to *B*, both nodes are still cut off.

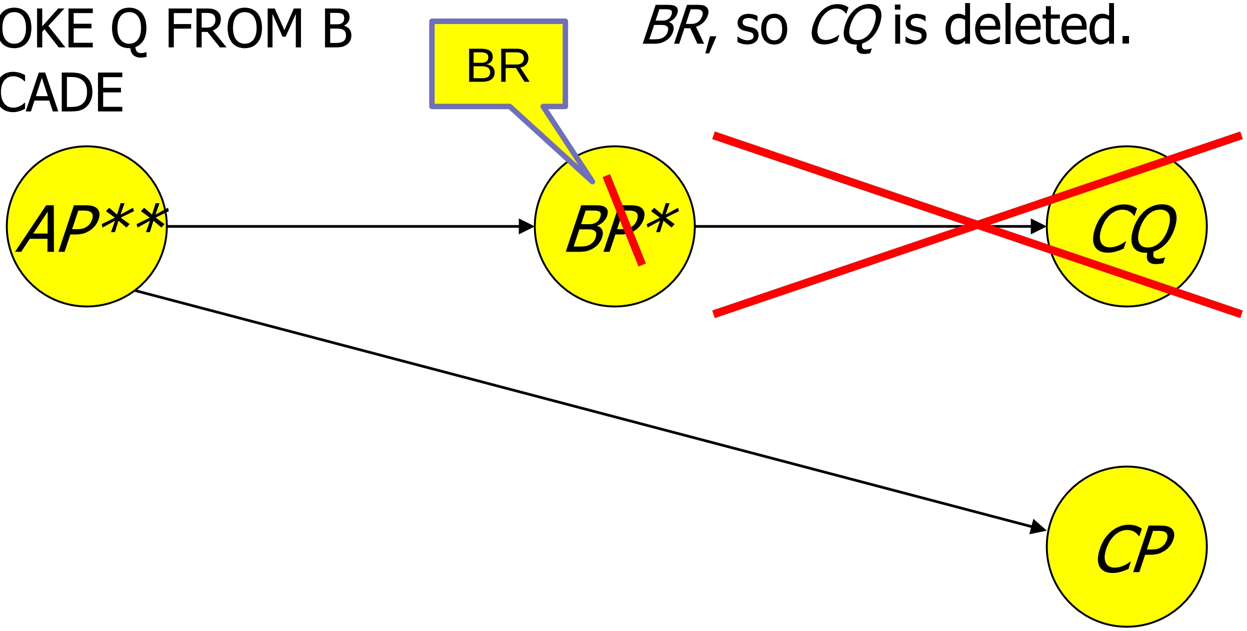
However, *C* still has *P* without grant option because of the direct grant.

Privileges – Diagram

P = {Q*,R}

A:
REVOKE Q FROM B
CASCADE

Not only *B* loses *Q*, but *BP** changes to *BR*, so *CQ* is deleted.



Contents

- Generating IDs
- Spatial data
  - Data types, indexing
- DB security
  - Access control in DB
  - **Stored procedures**
  - Attack on DB

Stored Procedures

- User-defined program implementing an activity
  - E.g., factorial computation, distance between GPS coords, inserting rows to multiple tables, …
- PostgreSQL
  - CREATE FUNCTION name ([parameters,…]) [RETURNS type]
    …code…

Stored Procedures

- Example:
  - Compute average salary without revealing the individual salaries
    - Table Employee(id, name, address, salary)
  - PostgreSQL:
    - CREATE FUNCTION avgsal() RETURNS real AS ‘SELECT avg(salary) FROM employee’ LANGUAGE SQL;
  - User executes the procedure (function):
    - SELECT avgsal();

Stored Procedures

- Example (cont.):
  - Salaries are not *secured*
  - To secure we need to
    - REVOKE SELECT ON Employee FROM …
    - GRANT EXECUTE ON FUNCTION avgsal() TO …
  - By running “SELECT avgsal();” the procedure is executed with privileges of current user.
  - → it needs SELECT on Employee!

Stored Procedures

- Context of execution
  - Can be set during procedure creation
  - Types:
    - **INVOKER** – run in the context of user that calls the function (typically current user)
    - **DEFINER**– run in the context of the owner of function
    - **„particular user“** – run in the context of the selected user
    - …

Stored Procedures

- Execution context
  - PostgreSQL
    - SECURITY INVOKER
    - SECURITY DEFINER
- Solution: set the context to owner
  - CREATE FUNCTION …. LANGUAGE SQL **SECURITY DEFINER**;
    - Assumption: owner has the SELECT privilege to Employee

Attacks to DB system

- Network connection
  - DB port open to anyone → use firewall
  - Unsecured connection
    - Apply SSL
- Logging in
  - Weak password
  - Limit users to logging in
    - Allow selected user accounts, IP addresses and databases
  - Using one generic (admin) DB account

Attacks to DB system

- SQL injection
  - Attack by sending SQL commands in place of valid data in forms.
  - Typically related to using only one DB account
    - which is admin)-:

SQL injection – example

- App presents a form to enter string to update customer’s note in DB:
  - □ Internally the app use the following DB statement:

```
UPDATE customer SET note=‘$note’
WHERE id=current_user;
```

- Malicious user enters to the form:

```
Vader’; DROP TABLE customer; --
```

- After variable expansion we get string:

```
UPDATE customer
SET note=’Vader’; DROP TABLE customer; --’
WHERE id=current_user;
```

All in one line!

SQL Injection: Countermeasures

- Use specific user account
  - Avoid using admin account
- Check input values
  - Input length, escape characters,…
- Functions in programming language
  - *mysql_real_escape_string()*, *add_slashes()*
  - $dbh->*quote*($string)
- Functions in DB
  - *quote_literal(str)*
    - returns a string *str* suitably quoted to be used as a string literal in an SQL statement

SQL Injection: Countermeasures

- Prepared statements
  - Parsed statements prepared in DB
    - i.e., compiled templates ready for use
  - Values are then substituted
    - Parameters do not need to be quoted then
  - May be used repetitively
  - Example:

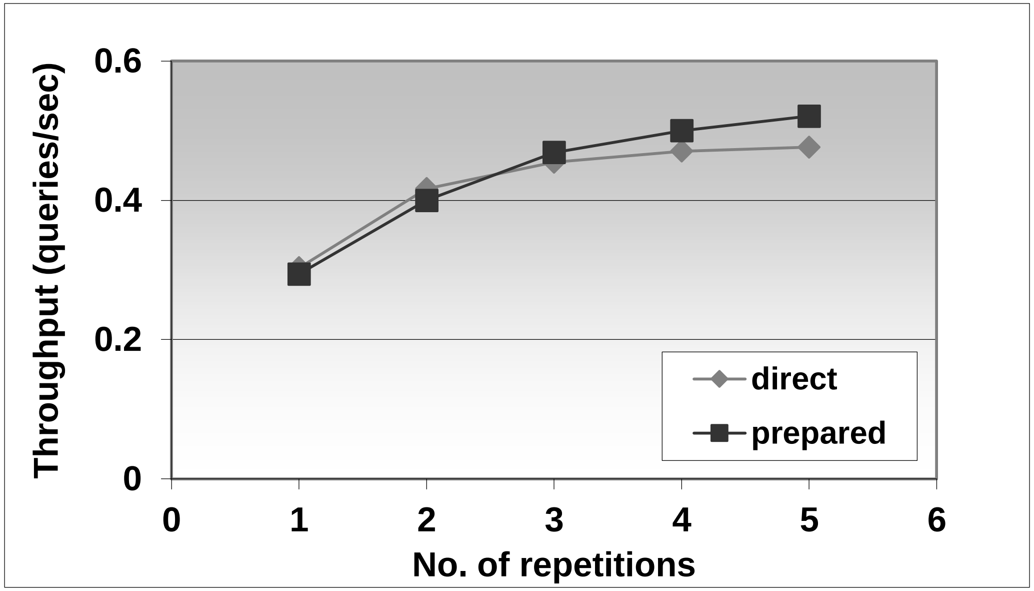
```
$st = $dbh->prepare("SELECT * FROM emp WHERE name LIKE ?");
$st->execute(array( "%$_GET[name]%“ ));
```

SQL Injection: Countermeasures

- Prepared statements at server-side
  - The same concept, but stored in DB
  - Typically in procedural languages in DB
  - PostgreSQL
    - PREPARE emp_row(text) AS SELECT * FROM emp
      WHERE name LIKE **$1**;
      EXECUTE emp_row(**‘%John%’**);
- Query is planned in advance
  - Planning time can be amortized
  - But: the plan is generic!
    - i.e., without any optimization induced by knowing the parameter
  - Lasts only for the duration of the current db session

Prepared Statements: Performance

- Prepared execution yields better performance when the query is executed more than once:
  - □ No compilation
  - □ No access to catalog.



- Experiment performed on Oracle8iEE on Windows 2000.

Attacking Views

- Views protect data rows…
  - if permissions are correctly set
  - E.g., student(studentid, firstname, lastname, fieldofstudy)

    -
      ```
      CREATE OR REPLACE VIEW studentssme AS  SELECT * FROM student
      WHERE fieldofstudy = 'N-SSME‘;
      ```
  - But, creating a “cheap” function

    -
      ```
      CREATE OR REPLACE FUNCTION test(name text, study text)
      RETURNS boolean AS $$
        begin
          raise notice 'Name: %, Study: %', name, study;
          return true;
        end;
      $$  LANGUAGE plpgsql VOLATILE  COST 0.00001;
      ```
  - The query leaks other students in a side channel…

    -
      ```
      SELECT * FROM studentssme WHERE test(lastname, fieldofstudy)
      ```

      -
        ```
        NOTICE:  Name: Nový, Study: N-AplInf
        NOTICE:  Name: Dlouhý, Study: N-Inf
        NOTICE:  Name: Svoboda, Study: N-AplInf
        NOTICE:  Name: Starý, Study: N-SSME
        NOTICE:  Name: Lukáš, Study: N-SSME
        …
        ```
- Countermeasures:
  - ban creating new DB objects
  - use security_barrier in Pg.conf or in create view

Lecture Takeaways

- Primary key value generation
- Extensions to more complex data with indexing support
- Securing DB
  - Avoid using admin account for general use
  - Mind “no-action” revoke command and recheck the resulting graph of grants.