

Lecture 10: Physical Database Design, Security

- Physical Database Design and Tuning
 - Workload
 - Decisions to be made
 - Heuristics
 - Tuning
 - Horizontal Decomposition
- Security & Authorization
 - Access Controls
 - Discretionary
 - Mandatory

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 1
Lecture 10

Physical Database Design and Tuning

- After ER design, schema refinement, and the definition of views, we have the *conceptual* and *external* schemas for our database.
- The next step is to choose indexes, make clustering decisions, and (maybe) refine the conceptual and external schemas to meet performance goals.
- Begins with the *workload*:
 - The most important queries and how often they arise.
 - The most important updates and how often they arise.
 - Performance targets.

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 2
Lecture 10

Decisions to Make

- What indexes should we create?
- What kind of an index?
 - Clustered? Hash/tree?
- Simple strategy:
 - Choose the most important queries and updates
 - Add whatever indexes speed up those queries
 - Worry about affect on indexes of updates
- Change the conceptual schema?
 - Alternative normalized schemas?
 - "Undo" some decomposition steps?
 - Horizontal partitioning?

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 3
Lecture 10

Example 1

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno
```

- Hash index on E.dno allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.
 - What plans could use this index?
- Hash index on D.dname supports 'Toy' selection.
 - What plans could use this hash index?
 - ~~Given this index on D.dno is not needed~~
 - Why?

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 4
Lecture 10

Example 1a

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname='Toy' AND E.dno=D.dno
AND E.age = 25
```

- What if WHERE included: "... AND E.age=25" ?
 - Could retrieve Emp tuples using index on E.age, then join with Dept tuples satisfying dname selection.
 - So, if E.age index is already created, is it worth adding an index on E.dno?

Example 2

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.sal BETWEEN 10000 AND 20000
AND E.hobby='Stamps' AND E.dno=D.dno
```

- What if we build a hash index on D.dno?
 - Which plans can use that index?
- What index should we build on Emp?
 - B+ tree on E.sal could be used, OR an index on E.hobby could be used. Only one of these is needed, and which is better depends upon the selectivity of the conditions.
 - As a rule of thumb, equality selections more selective than range selections.
- As both examples indicate, our choice of indexes is guided by the plan(s) that we expect an optimizer to consider for a query. Have to understand optimizers!

Database Tuning

- Database tuning: modifying the physical and conceptual schemas to meet changing needs
- First, index tuning. Why do we do that?
 - Workloads change
 - The optimizer may not behave as planned
 - Statistics may need to be refreshed
 - Static indexes (ISAM, static hash) may need to be refreshed
- Next, conceptual schema tuning

Schema Tuning

- The choice of conceptual schema should be guided by the workload, in addition to redundancy issues:
 - We may settle for a 3NF schema rather than BCNF.
 - Workload may influence the choice we make in decomposing a relation into 3NF or BCNF.
 - We may further decompose a BCNF schema!
 - We might *denormalize* (i.e., undo a decomposition step), or we might add fields to a relation.
 - We might consider *horizontal decompositions*.

Example Schemas

Contracts (<u>Cid</u> , Sid, Jid, Did, Pid, Qty, Val)
Depts (<u>Did</u> , Budget, Report)
Suppliers (<u>Sid</u> , Address)
Parts (<u>Pid</u> , Cost)
Projects (<u>Jid</u> , Mgr)

- We will concentrate on *Contracts*, denoted as *CSJDPQV*. The following FDs are given to hold: $JP \rightarrow C$, $SD \rightarrow P$, C is the primary key.
 - What are the keys for *CSJDPQV*?
 - In what normal form is this relation schema?

Settling for 3NF vs BCNF

- *CSJDPQV* can be decomposed into *SDP* and *CSJDQV*, and both relations are in *BCNF*. (Which FD suggests that we do this?)
 - Lossless decomposition, but not dependency-preserving.
 - Adding *CJP* makes it dependency-preserving as well.
- Suppose that this query is very important:
 - Find the quantity *Q* of part *P* ordered in contract *C*.
 - Requires a join on the decomposed schema, but can be answered by a scan of the original relation *CSJDPQV*.
 - Could lead us to settle for the 3NF schema *CSJDPQV*.

Denormalization

- Suppose that the following query is important:
 - *Is the value of a contract less than the budget of the department?*
- To speed up this query, we might add a field budget B to Contracts.
 - *This introduces the FD $D \rightarrow B$ in Contracts.*
 - *Thus, Contracts is no longer in 3NF.*
- We might choose to modify Contracts this way if the query is sufficiently important, and we cannot obtain adequate performance otherwise (i.e., by adding indexes or by choosing an alternative 3NF schema).

Choice of Decompositions

- There are 2 ways to decompose CSJDPQV into BCNF:
 - SDP and CSJDQV; lossless-join but not dep-preserving.
 - SDP, CSJDQV and CJP; dep-preserving as well.
- The difference between these is really the cost of enforcing the FD $JP \rightarrow C$ versus the cost of supporting one more relation.
 - 2nd decomposition: Declare JP to be a key in relation CJP.
 - 1st:

```
CREATE ASSERTION CheckDep
CHECK ( NOT EXISTS ( SELECT *
FROM PartInfo P, ContractInfo C
WHERE P.sid=C.sid AND P.did=C.did
GROUP BY C.jid, P.pid
HAVING COUNT (C.cid) > 1 ))
```

Horizontal Decompositions

- Our definition of decomposition: Relation is replaced by a collection of relations that are *projections*. Most important case.
- Sometimes, might want to replace relation by a collection of relations that are selections.
 - Each new relation has same schema as the original, but a subset of the rows.
 - Collectively, new relations contain all rows of the original. Typically, the new relations don't overlap.

Horizontal Decompositions (Contd.)

- Suppose that contracts with value > 10000 are subject to different rules. This means that queries on *Contracts* will often contain the condition $val > 10000$.
- One way to deal with this is to build a clustered B+ tree index on the *val* field of *Contracts*.
- A second approach is to replace contracts by two new relations: *LargeContracts* and *SmallContracts*, with the same attributes (CSJDPQV).
 - Performs like index on such queries, but no index overhead.
 - Can build clustered indexes on other attributes, in addition!

Masking Conceptual Schema Changes

```
CREATE VIEW Contracts(cid, sid, jid, did, pid, qty, val)
AS SELECT *
FROM LargeContracts
UNION
SELECT *
FROM SmallContracts
```

- The replacement of *Contracts* by *LargeContracts* and *SmallContracts* can be masked by the view.
- However, queries with the condition $val > 10000$ must be asked on *LargeContracts* for efficient execution: Users concerned with performance have to be aware of the change.

Tuning Queries and Views

- If a query runs slower than expected, check if an index needs to be re-built, or if statistics are too old.
- Sometimes, the DBMS may not be executing the plan you had in mind. Common areas of weakness:
 - Selections involving *null values*.
 - Selections involving *arithmetic or string expressions*.
 - Selections involving *OR* conditions.
 - *Lack of evaluation features* such certain join methods.
- Check the plan that is being used! Then adjust the choice of indexes or *rewrite the query/view*.

Rewriting SQL Queries

- Guideline: Use only one "query block", if possible.

```

SELECT DISTINCT *
  FROM Sailors S
 WHERE S.sname IN
       (SELECT Y.sname
        FROM YoungSailors Y)
    =
SELECT DISTINCT S.*
  FROM Sailors S,
       YoungSailors Y
 WHERE S.sname = Y.sname
    
```

More Guidelines for Query Tuning

- Minimize the use of DISTINCT: don't need it if duplicates are acceptable, or if answer contains a key.
- Minimize the use of GROUP BY and HAVING:

```

SELECT MIN (E.age)
FROM Employee E
GROUP BY E.dno
HAVING E.dno=102
    
```

```

SELECT MIN (E.age)
FROM Employee E
WHERE E.dno=102
    
```

- Consider DBMS use of index when writing arithmetic expressions: $E.age = 2 * D.age$ will benefit from index on $E.age$, but might not benefit from index on $D.age$!

Guidelines for Query Tuning (Contd.)

- Avoid using intermediate relations:

```
SELECT * INTO TempLen
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname='Joe'
```

```
SELECT E.dno, AVG(E.sal)
FROM Emp E, Dept D
WHERE E.dno=D.dno
      AND D.mgrname='Joe'
GROUP BY E.dno
```

vs.

and

```
SELECT T.dno, AVG(T.sal)
FROM TempLen T
GROUP BY T.dno
```

Security and Authorization, Chapter 21

- Objectives:
 - **Secrecy:** Users should not be able to see things they are not supposed to.
 - E.g., A student can't see other students' grades.
 - **Integrity:** Users should not be able to modify things they are not supposed to.
 - E.g., Only instructors can assign grades.
 - **Availability:** Users should be able to see and modify things they are allowed to.
- OS, Network security issues also relevant
 - Only database issues discussed here

Access Controls

- A **security policy** specifies who is authorized to do what.
- A **security mechanism** allows us to enforce a chosen security policy.
- Two main mechanisms at the DBMS level:
 - Discretionary access control
 - Mandatory access control

Discretionary Access Control

- Based on the concept of access rights or **privileges** for objects (tables and views), and mechanisms for giving users privileges (and revoking privileges).
- Creator of a table automatically gets all privileges on it.
 - DMBS tracks who subsequently gains and loses privileges, and only allows requests from users who have the necessary privileges.

GRANT Command

`GRANT privileges ON object TO auth [WITH GRANT OPTION]`

- The following **privileges** can be specified:
 - ❖ **SELECT**: Can read all columns.
 - ❖ **INSERT**: Can insert tuples.
 - ❖ **DELETE**: Can delete tuples.
 - ❖ **REFERENCES (col-name)**: Can define foreign keys (in other tables) that refer to this column.
- **Auth** is a group of users.
- If a user has a privilege with the **GRANT OPTION**, can pass privilege on to other users (with or without passing on the **GRANT OPTION**).
- Only owner can execute **CREATE**, **ALTER**, and **DROP**.

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 23
Lecture 10

REVOKE Command

`REVOKE privileges ON object FROM auth [RESTRICT|CASCADE]`

- **Must be issued by a user who granted "privileges" on "object" to "auth".**
- **CASCADE:**
 - *abandon* anyone granted this privilege by auth, and so on recursively.
 - If an auth's privilege is abandoned and this is the last possible abandonment, then revoke auth's privilege
 - "and so on recursively": Abandon anyone granted the privilege by the auth just abandoned.
- **RESTRICT: If CASCADE would cause anyone to be abandoned, then cancel the command.**

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 24
Lecture 10

Examples of Grant, Revoke

- Suppose Len has created the author table, then these commands are executed in this order
 - Len: GRANT UPDATE, SELECT ON *agents* TO Yingjin
 - Len: GRANT UPDATE ON *agents* TO Parisa WITH GRANT OPTION
 - Parisa: GRANT UPDATE ON *agents* TO Yingjin
 - Len: REVOKE UPDATE ON *agents* FROM Parisa CASCADE
 - Len: REVOKE SELECT ON *agents* FROM Yingjin CASCADE
- What privileges does Parisa have on *agents*?
- What privileges does Yingjin have on *agents*?
- Can Yingjin issue **UPDATE *agents* A SET A.age=25?**
- Can Yingjin issue **UPDATE *agents* A SET A.age=A.age+1?**

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 25
Lecture 10

REVOKE more precisely

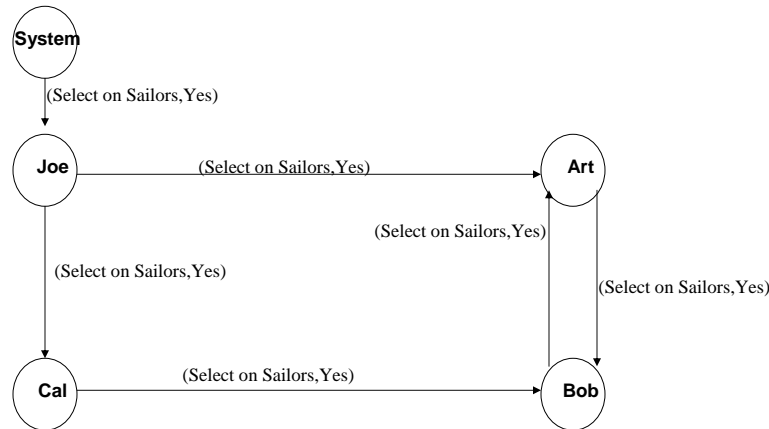
Steps to be taken after "REVOKE P ON O FROM A"

- First consider the case *CASCADE*
 1. Draw an authorization graph of current state (next slide)
 2. Delete the arc corresponding to "GRANT P ON O TO A"
 3. Also delete any arc not accessible from *SYSTEM* via grants of the privilege P on O
 4. If an auth has no incoming arcs granting P on O, then auth loses the privilege P on O.
- *RESTRICT* is the same as before: If *CASCADE* would cause any arc other than "GRANT ON P TO A" to be deleted, then abort the command.

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 26
Lecture 10

Authorization Graph



CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 27
Lecture 10

Role-Based Authorization

- In SQL-92, privileges are actually assigned to **authorization ids**, which can denote a single user or a group of users.
- In SQL:1999 privileges are assigned to **roles**.
 - Roles can then be granted to users and to other roles.
 - Reflects how real organizations work.

CS386 Introduction to Database Systems, © Lois Delcambre, David Maier 1999-2006
Some slides adapted from R. Ramakrishnan, with permission

Slide 28
Lecture 10

Mandatory Access Control

- Based on system-wide policies that cannot be changed by individual users.
 - Each **DB object** is assigned a **security class**.
 - Each **subject** (user or user program) is assigned a **clearance** for a security class.
 - Rules based on security classes and clearances govern who can read/write which objects.
- Most commercial systems do not support mandatory access control. Versions of some DBMSs do support it; used for specialized (e.g., military) applications.

Why Mandatory Control?

- Discretionary control has some flaws, e.g., the *Trojan horse* problem:
 - Justin has access to secret information that Dick wants to know.
 - Dick creates a table called *Secret* and gives INSERT privileges to Justin (who doesn't know about this).
 - Dick modifies the code of a database application program used by Justin to additionally write some secret data to table *Secret*.
 - This requires hacking the OS security system a bit.
 - Now, Dick can see the secret data.
- The modification of the code is beyond the DBMSs control, but it can try and prevent the use of the database as a **channel** for secret information.

Bell-LaPadula Model

- Objects (e.g., tables, views, tuples)
- Subjects (e.g., users, user programs)
- Security classes:
 - Top secret (TS), secret (S), confidential (C), unclassified (U): $TS > S > C > U$
- Each object and subject is assigned a class.
 - Subject S can read object O only if $class(S) \geq class(O)$ (Simple Security Property)
 - Subject S can write object O only if $class(S) \leq class(O)$ (*-Property)

Intuition

- Never let information flow from a higher to a lower security level.
- E.g., If Dick is cleared for security class C , Justin for class S , and the secret information has class S :
 - Dick's table, Secret, has Dick's clearance, C .
 - Justin's application has his clearance, S .
 - So, the database application cannot write into table Secret.
- The mandatory access control rules are applied in addition to any discretionary controls that are in effect.

Multilevel Relations

<u>bid</u>	bname	color	class
101	Salsa	Red	S
102	Pinto	Brown	C

- Users with *S* and *TS* clearance will see both rows; a user with *C* will only see the 2nd row; a user with *U* will see no rows.
- If user with *C* tries to insert $\langle 101, \text{Pasta}, \text{Blue}, C \rangle$:
 - Allowing insertion violates key constraint
 - Disallowing insertion tells user that there is another object with key 101 that has a class $\succ C$!
 - Fix by treating class field as part of key.