CS 458 / 658: Computer Security and Privacy

Module 6 - Data Security and Privacy

Part 1 - On the security of databases

Meng Xu *(University of Waterloo)*

Winter 2023

Outline

1. Background: relational database
2. Access control
3. Integrity
4. Others

Relational Databases

Q: What is a relational database?

Relational Databases

Q: What is a relational database?

A: A relational database is a structured collection of data (records) following a relational model.

Relational Databases

Q: What is a relational database?

A: A relational database is a structured collection of data (records) following a relational model.

A relational model

- Table has rows (records) and named columns (attributes).
- Tables can be related to one another (hence, relations).

Relational Databases

Q: What is a relational database?

A: A relational database is a structured collection of data (records) following a relational model.

A relational model

- Table has rows (records) and named columns (attributes).
- Tables can be related to one another (hence, relations).

The relational model, sometimes also referred to as the schema, is usually set by database administrator

Relational Databases

Q: What is a relational database?

A: A relational database is a structured collection of data (records) following a relational model.

A relational model

- Table has rows (records) and named columns (attributes).
- Tables can be related to one another (hence, relations).

The relational model, sometimes also referred to as the schema, is usually set by database administrator

Database management system (DBMS) provides support for queries and management of the records.

Relations: example

Here is a table that an airline booking agency might use to store details of their customers:

Last	**First**	**Address**	**City**	**State**	**Zip**	**Airport**
ADAMS	Charles	212 Market St.	Columbus	OH	43210	CMH
ADAMS	Edward	212 Market St.	Columbus	OH	43210	CMH
BENCHLY	Zeke	501 Union St.	Chicago	IL	60603	ORD
CARTER	Marlene	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Beth	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Ben	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Lisabeth	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Mary	411 Elm St.	Columbus	OH	43210	CMH

Relations: example

Here is a table that an airline booking agency might use to store details of their customers:

Last	First	Address	City	State	Zip	Airport
ADAMS	Charles	212 Market St.	Columbus	OH	43210	CMH
ADAMS	Edward	212 Market St.	Columbus	OH	43210	CMH
BENCHLY	Zeke	501 Union St.	Chicago	IL	60603	ORD
CARTER	Marlene	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Beth	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Ben	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Lisabeth	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Mary	411 Elm St.	Columbus	OH	43210	CMH

Q: What is the issue with storing data in a flattened table like this?

Relations: example

Here is a table that an airline booking agency might use to store details of their customers:

Last	First	Address	City	State	Zip	Airport
ADAMS	Charles	212 Market St.	Columbus	OH	43210	CMH
ADAMS	Edward	212 Market St.	Columbus	OH	43210	CMH
BENCHLY	Zeke	501 Union St.	Chicago	IL	60603	ORD
CARTER	Marlene	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Beth	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Ben	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Lisabeth	411 Elm St.	Columbus	OH	43210	CMH
CARTER	Mary	411 Elm St.	Columbus	OH	43210	CMH

Q: What is the issue with storing data in a flattened table like this?

A: Lots of repeated values. This affects the storage cost, query speed, difficulty of maintenance, etc

Relations: normalization

Table: FamilyInfo

Last	**Address**	**City**	**State**	**Zip**
ADAMS	212 Market St.	Columbus	OH	43210
BENCHLY	501 Union St.	Chicago	IL	60603
CARTER	411 Elm St.	Columbus	OH	43210

Last	**First**
ADAMS	Charles
ADAMS	Edward
BENCHLY	Zeke
CARTER	Marlene
CARTER	Beth
CARTER	Ben
CARTER	Lisabeth
CARTER	Mary

Table: NameInfo

Zip	**Airport**
43210	CMH
60603	ORD

Table: AirportInfo

Relations: normalization

Normalization eliminates redundant storage of data, which

- optimizes the storage costs,
- improves query speed, and
- reduces future maintenance costs.

Database queries

The most popular language for query and manipulation of a relational database is SQL.

Database queries

The most popular language for query and manipulation of a relational database is SQL.

- A single table query

```
SELECT Address FROM FamilyInfo
 WHERE (Zip = "43210") AND (Name ="ADAMS")
```

Database queries

The most popular language for query and manipulation of a relational database is SQL.

- A single table query

```
SELECT Address FROM FamilyInfo
 WHERE (Zip = "43210") AND (Name ="ADAMS")
```

- A join query across multiple tables

```
SELECT Name, Airport
 FROM FamilyInfo JOIN AirportInfo
 ON FamilyInfo.Zip = AirportInfo.Zip
```

Database queries

The most popular language for query and manipulation of a relational database is SQL.

- A single table query

```
SELECT Address FROM FamilyInfo
 WHERE (Zip = "43210") AND (Name ="ADAMS")
```

- A join query across multiple tables

```
SELECT Name, Airport
 FROM FamilyInfo JOIN AirportInfo
 ON FamilyInfo.Zip = AirportInfo.Zip
```

- An aggregation

```
SELECT COUNT(Last) FROM FamilyInfo
 WHERE City = "Columbus"
```

Database queries

The most popular language for query and manipulation of a relational database is SQL.

- A single table query

```
SELECT Address FROM FamilyInfo
 WHERE (Zip = "43210") AND (Name ="ADAMS")
```

- A join query across multiple tables

```
SELECT Name, Airport
 FROM FamilyInfo JOIN AirportInfo
 ON FamilyInfo.Zip = AirportInfo.Zip
```

- An aggregation

```
SELECT COUNT(Last) FROM FamilyInfo
 WHERE City = "Columbus"
```

- A change of record content

```
UPDATE FamilyInfo SET Address = "1 Town St."
 WHERE Last = "ADAMS"
```

Security requirements for a database

Security requirements for a database

- Access control
    - who can read? who can write?

Security requirements for a database

- Access control
  - who can read? who can write?
- Authentication
  - how do we know if a DB client is not masquerading as someone else

Security requirements for a database

- Access control
  - who can read? who can write?
- Authentication
  - how do we know if a DB client is not masquerading as someone else
- Confidentiality
  - what if the DB server is compromised? what about network tapping?

Security requirements for a database

- Access control
  - who can read? who can write?
- Authentication
  - how do we know if a DB client is not masquerading as someone else
- Confidentiality
  - what if the DB server is compromised? what about network tapping?
- Integrity
  - how do we guarantee that the data is in an intact and sensible state

Security requirements for a database

- Access control
  - who can read? who can write?
- Authentication
  - how do we know if a DB client is not masquerading as someone else
- Confidentiality
  - what if the DB server is compromised? what about network tapping?
- Integrity
  - how do we guarantee that the data is in an intact and sensible state
- Availability
  - redundancy? fault-tolerance? Byzantine fault tolerance?

Security requirements for a database

- Access control
    - who can read? who can write?
- Authentication
    - how do we know if a DB client is not masquerading as someone else
- Confidentiality
    - what if the DB server is compromised? what about network tapping?
- Integrity
    - how do we guarantee that the data is in an intact and sensible state
- Availability
    - redundancy? fault-tolerance? Byzantine fault tolerance?
- Auditability
    - a.k.a. provenance, proving how we ended up with a specific state

Security requirements for a database

- Access control
  - who can read? who can write?
- Authentication
  - how do we know if a DB client is not masquerading as someone else
- Confidentiality
  - what if the DB server is compromised? what about network tapping?
- Integrity
  - how do we guarantee that the data is in an intact and sensible state
- Availability
  - redundancy? fault-tolerance? Byzantine fault tolerance?
- Auditability
  - a.k.a. provenance, proving how we ended up with a specific state

Outline

1. Background: relational database
2. Access control
3. Integrity
4. Others

Access control - Recall OS module

Q: What are the access control models you have learned?

Access control - Recall OS module

Q: What are the access control models you have learned?

A: DAC, RBAC, MAC

Access control - Recall OS module

Q: What are the access control models you have learned?

A: DAC, RBAC, MAC

- Discretionary Access Control (DAC)
  - owners can delegate (grant/revoke) privileges to others
- Role-based Access Control (RBAC)
  - ties in users' privileges to their position or roles in the organization
- Mandatory Access Control (MAC)
  - users and objects are assigned labels based on their 'security level'

Access control - Recall OS module

Q: What are the access control models you have learned?

A: DAC, RBAC, MAC

- Discretionary Access Control (DAC)
  - owners can delegate (grant/revoke) privileges to others
  - *If you own the data, you can do anything with it.*
- Role-based Access Control (RBAC)
  - ties in users' privileges to their position or roles in the organization
  - *Assign labels to users and assign privileges to labels.*
- Mandatory Access Control (MAC)
  - users and objects are assigned labels based on their 'security level'
  - *You don't own the data even if you create it. The data has labels too and may deny access from its creator.*

Access control - Recall OS module

Q: What are the access control models you have learned?

A: DAC, RBAC, MAC

- Discretionary Access Control (DAC)
  - owners can delegate (grant/revoke) privileges to others
  - *If you own the data, you can do anything with it.*
- Role-based Access Control (RBAC)
  - ties in users' privileges to their position or roles in the organization
  - *Assign labels to users and assign privileges to labels.*
- Mandatory Access Control (MAC)
  - users and objects are assigned labels based on their 'security level'
  - *You don't own the data even if you create it. The data has labels too and may deny access from its creator.*

Access control for databases

All three types of access control (DAC, RBAC, MAC) apply to databases (with various forms of implementations).

Access control for databases

All three types of access control (DAC, RBAC, MAC) apply to databases (with various forms of implementations).

- Most commercial DBs have native support for DAC and RBAC
- Multi-level security database is an implementation of MAC

Access control for databases

All three types of access control (DAC, RBAC, MAC) apply to databases (with various forms of implementations).

- Most commercial DBs have native support for DAC and RBAC
- Multi-level security database is an implementation of MAC

Q: What is the design space of a database access control scheme (i.e., what is the data and what are the privileges?)

Access control for databases

All three types of access control (DAC, RBAC, MAC) apply to databases (with various forms of implementations).

- Most commercial DBs have native support for DAC and RBAC
- Multi-level security database is an implementation of MAC

Q: What is the design space of a database access control scheme (i.e., what is the data and what are the privileges?)

A:

- Granularity of data: access control on *relations*, *records*, *attributes*
- Supporting different operations: `SELECT`, `INSERT`, `UPDATE`, `DELETE`

DAC for databases

DAC is built into the SQL language.

DAC for databases

DAC is built into the SQL language.

- Use the `GRANT` keyword to assign a privilege to a user
- Use the `REVOKE` keyword to withdraw a privilege.

DAC for databases

DAC is built into the SQL language.

- Use the `GRANT` keyword to assign a privilege to a user
- Use the `REVOKE` keyword to withdraw a privilege.

Different types of privileges have built-in support:

- Account-level privileges:
  - DBMS functionalities (e.g. shutdown server),
  - creating or modifying tables,
  - routines (database functions),
  - users and roles.
- Relation-level privileges:
  - `SELECT`,
  - `UPDATE`,
  - `REFERENCES` privileges on a relation

DAC example: account-level privilege

Accounts A1, A2
Relations: nil

Account-level privilege

```
> Admin:  GRANT CREATE USER TO A1;
```

Sysadmin grants user A1 privilege to create users (and roles).

DAC example: account-level privilege

Accounts A1, A2 , A3
Relations: nil

Account-level privilege

```
> Admin:  GRANT CREATE USER TO A1;
```

Sysadmin grants user A1 privilege to create users (and roles).

Account-level privilege

```
> A1:  CREATE USER A3;
```

User A1 now uses her privilege to create another user.

DAC example: account-level privilege

Accounts A1, A2, A3
Relations: nil

Account-level privilege

```
> Admin:  GRANT CREATE TABLE TO A2;
```

Sysadmin grants user A2 privilege to create new tables.

DAC example: account-level privilege

Accounts A1, A2, A3

Relations: Employee

Account-level privilege

```
> Admin: GRANT CREATE TABLE TO A2;
```

Sysadmin grants user A2 privilege to create new tables.

Account-level privilege

```
> A2: CREATE TABLE Employee (...);
```

User A2 now uses her privilege to create the `Employee` table.

DAC example: relation-level privilege

Accounts A1, A2, A3
Relations: `Employee`

Relation-level privilege

```
> A2:  GRANT SELECT ON Employee TO A3;
```

The table owner (A2) grants user A3 the privilege to run `SELECT` queries on the `Employee` table.

DAC example: relation-level privilege

Accounts A1, A2, A3
Relations: `Employee`

Relation-level privilege

```
> A2:  GRANT SELECT ON Employee TO A3;
```

The table owner (A2) grants user A3 the privilege to run `SELECT` queries on the `Employee` table.

Relation-level privilege

```
> A2:  GRANT SELECT ON Employee TO A3 WITH GRANT OPTION;
```

The table owner (A2) grants user A3 the privilege to run `SELECT` queries on the `Employee` table and to further delegate that privilege to other users.

DAC example: relation-level privilege

Accounts A1, A2, A3

Relations: `Employee`

Relation-level privilege

```
> A3:  GRANT SELECT ON Employee TO A1;
```

A3 now can exercise her delegation rights

DAC example: relation-level privilege

Accounts A1, A2, A3
Relations: `Employee`

Relation-level privilege

```
> A3:  GRANT SELECT ON Employee TO A1;
```

A3 now can exercise her delegation rights

Relation-level privilege

```
> A2:  REVOKE SELECT ON Employee FROM A1;
```

The table owner (A2) however, reserves the rights to revoke any privilege she considers as improper.

Fine-grained DAC

Q: What is missing in the DAC scheme we have seen so far?

Fine-grained DAC

Q: What is missing in the DAC scheme we have seen so far?

A:

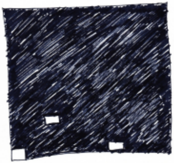


Fig. 74. "Privacy means my life is a black box, except for the items I choose to share with others." By Lauren, age 32

Fine-grained DAC

Q: What is missing in the DAC scheme we have seen so far?

A:

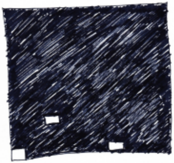


Fig. 74. "Privacy means my life is a black box, except for the items I choose to share with others." By Lauren, age 32

The solution is SQL views:

- For an SQL query, we can generate a view that represents the result of that query.
- Views can be used to only reveal certain columns (attributes after `SELECT`) and rows (defined by the `WHERE` clause) for access control.

Fine-grained DAC using SQL views

Accounts A1, A2, A3

Relations: `Employee(Name, SIN, DOB, Address, Salary, Dpt)`

Create a view

```
> A2:  CREATE VIEW CSEmployeePublicInfo
             SELECT Name, DOB, Address FROM Employee
             WHERE Dpt = "CS";
```

The table owner (A2) creates a view that only expose the (`Name`, `DOB`, `Address`) information for `Employees` in the CS department.

Fine-grained DAC using SQL views

Accounts A1, A2, A3

Relations: `Employee(Name, SIN, DOB, Address, Salary, Dpt)`

Create a view

```
> A2:  CREATE VIEW CSEmployeePublicInfo
           SELECT Name, DOB, Address FROM Employee
           WHERE Dpt = "CS";
```

The table owner (A2) creates a view that only expose the (`Name`, `DOB`, `Address`) information for `Employees` in the CS department.

Relation-level privilege via views

```
> A2:  GRANT SELECT ON CSEmployeePublicInfo TO A3;
```

The table owner (A2) grants user A3 the privilege to run `SELECT` queries on the restrict view instead of the whole `Employee` table.

Fine-grained DAC: what about write operations?

Accounts A1, A2, A3

Relations: `Employee(Name, SIN, DOB, Address, Salary, Dpt)`

Fine-grained DAC: what about write operations?

Accounts A1, A2, A3

Relations: `Employee(Name, SIN, DOB, Address, Salary, Dpt)`

Column-specific update privilege

```
> A2:  GRANT UPDATE ON Employee (Address) TO A3;
```

The table owner (A2) grants user A3 the privilege to `UPDATE` the `Employee` table but only on the `Address` attribute.

Fine-grained DAC: what about write operations?

Accounts A1, A2, A3

Relations: `Employee(Name, SIN, DOB, Address, Salary, Dpt)`

Column-specific update privilege

```
> A2:  GRANT UPDATE ON Employee (Address) TO A3;
```

The table owner (A2) grants user A3 the privilege to `UPDATE` the `Employee` table but only on the `Address` attribute.

Q: How to restrict the `UPDATE` to selective rows only?

Fine-grained DAC: what about write operations?

Accounts A1, A2, A3

Relations: `Employee(Name, SIN, DOB, Address, Salary, Dpt)`

Column-specific update privilege

```
> A2:  GRANT UPDATE ON Employee (Address) TO A3;
```

The table owner (A2) grants user A3 the privilege to `UPDATE` the `Employee` table but only on the `Address` attribute.

Q: How to restrict the `UPDATE` to selective rows only?

A: Use `UPDATE` triggers (we will see this later)

From DAC to RBAC

Q: We already have DAC in SQL, why do we still need RBAC?

From DAC to RBAC

Q: We already have DAC in SQL, why do we still need RBAC?

A:

- DAC requires users to implement the principle of least privilege (hardly done in practice). Can lead to privilege escalation.
- System administrator needs to know how privileges are inter-related and assign multiple privileges for a user's tasks.
- Need to manually change privileges for multiple users who want to perform the same task, or when a user changes positions in an organization (i.e., roles).

RBAC for databases

Creating and using roles

```
> Admin:  CREATE ROLE "DptAdmin", "CompanyHR";
```

RBAC for databases

Creating and using roles

```
> Admin:  CREATE ROLE "DptAdmin", "CompanyHR";

> Admin:  GRANT "DptAdmin" TO A1;
> Admin:  GRANT "CompanyHR" TO A3;
```

RBAC for databases

Creating and using roles

```
> Admin:  CREATE ROLE "DptAdmin", "CompanyHR";

> Admin:  GRANT "DptAdmin" TO A1;
> Admin:  GRANT "CompanyHR" TO A3;

> A2:  GRANT SELECT ON CSEmployeePublicInfo TO "DptAdmin";
> A2:  GRANT UPDATE ON Employee(Address) TO "CompanyHR";
```

What about MAC?

We show a case study that aims to implement MAC for a database: multi-level security (MLS).

What about MAC?

We show a case study that aims to implement MAC for a database: multi-level security (MLS).

The theory behind MLS is the Bell-La Padula confidentiality model:

- There are security classifications or security levels applied to
  - *Subjects*: i.e., database users — security clearances
  - *Objects*: i.e., each cell in a table — security classifications

What about MAC?

We show a case study that aims to implement MAC for a database: multi-level security (MLS).

The theory behind MLS is the Bell-La Padula confidentiality model:

- There are security classifications or security levels applied to
  - *Subjects*: i.e., database users — security clearances
  - *Objects*: i.e., each cell in a table — security classifications
- An example of security levels:
  Top Secret > Secret > Classified > Unclassified

What about MAC?

We show a case study that aims to implement MAC for a database: multi-level security (MLS).

The theory behind MLS is the Bell-La Padula confidentiality model:

- There are security classifications or security levels applied to
  - *Subjects*: i.e., database users — security clearances
  - *Objects*: i.e., each cell in a table — security classifications
- An example of security levels:
  Top Secret $>$ Secret $>$ Classified $>$ Unclassified
- Security goal: ensures that information does not flow to those not cleared for that level.

What about MAC?

We show a case study that aims to implement MAC for a database: multi-level security (MLS).

The theory behind MLS is the Bell-La Padula confidentiality model:

- There are security classifications or security levels applied to
  - *Subjects*: i.e., database users — security clearances
  - *Objects*: i.e., each cell in a table — security classifications
- An example of security levels:
  Top Secret $>$ Secret $>$ Classified $>$ Unclassified
- Security goal: ensures that information does not flow to those not cleared for that level.
- Principles (simplified view):
  - *The simple security property*: S can read O iff $L(S) \geq L(O)$.
  - *The star property*: S can write O iff $L(S) \leq L(O)$.

Recall: Bell-LaPadula

Principles:

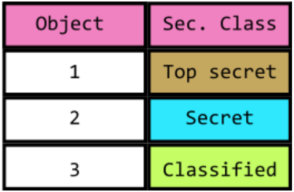
- *The simple security property*: S can read O iff $L(S) \geq L(O)$ (no read up)
- *The star property*: S can write O iff $L(S) \leq L(O)$ (no write down)

Q: Who can read what? Who can write what?

`Alice: Secret`

`Trent: Top secret`

`Bob: Classified`



Object	Sec. Class
1	Top secret
2	Secret
3	Classified

Recall: Bell-LaPadula

Principles:

- *The simple security property*: S can read O iff $L(S) \geq L(O)$ (no read up)
- *The star property*: S can write O iff $L(S) \leq L(O)$ (no write down)

Q: Who can read what? Who can write what?


Alice: Secret


Trent: Top secret


Bob: Classified

Object	Sec. Class	Can read?
1	Top secret	
2	Secret	
3	Classified	

Recall: Bell-LaPadula

Principles:

- *The simple security property*: S can read O iff $L(S) \geq L(O)$ (no read up)
- *The star property*: S can write O iff $L(S) \leq L(O)$ (no write down)

Q: Who can read what? Who can write what?


`Alice: Secret`


`Trent: Top secret`


`Bob: Classified`

Object	Sec. Class	Can read?	Can write?
1	Top secret		
2	Secret		
3	Classified		

MLS table example

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

MLS table example

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

- Each attribute has a classification label and a value at that label.

MLS table example

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

- Each attribute has a classification label and a value at that label.
- TC label = *Highest* clearance for any of its attributes.
  - TC: Tuple Classification

MLS table example

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

- Each attribute has a classification label and a value at that label.
- TC label = *Highest* clearance for any of its attributes.
  - TC: Tuple Classification
- Primary key label $\leq$ *Lowest* clearance for any of its attributes.
  - **Name** is the primary key in this example

MLS table example

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

- Each attribute has a classification label and a value at that label.
- TC label $=$ *Highest* clearance for any of its attributes.
  - TC: Tuple Classification
- Primary key label $\leq$ *Lowest* clearance for any of its attributes.
  - **Name** is the primary key in this example

Q: Why having this requirement?

MLS table example

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

- Each attribute has a classification label and a value at that label.
- TC label = *Highest* clearance for any of its attributes.
  - TC: Tuple Classification
- Primary key label $\leq$ *Lowest* clearance for any of its attributes.
  - **Name** is the primary key in this example

Q: Why having this requirement?

A: Otherwise a user may see a partial record without knowing what that record is about.

MLS read-down by filtering

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

MLS read-down by filtering

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

Filtering the table for users having classified clearance:

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	-	C	C
Brown	C	-	C	Good	C	C

MLS read-down by filtering

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

Filtering the table for users having classified clearance:

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	-	C	C
Brown	C	-	C	Good	C	C

Filtering the table for users having unclassified clearance:

Name		**Salary**		**Perf**		**TC**
Smith	U	-	U	-	U	U

MLS invisible polyinstantiation

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with classified clearance issues a write-up:

```
UPDATE Employee SET Perf = "Great" WHERE Name = "Smith";
```

MLS invisible polyinstantiation

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with classified clearance issues a write-up:

```
UPDATE Employee SET Perf = "Great" WHERE Name = "Smith";
```

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Smith	U	40000	C	Great	C	C
Brown	C	80000	S	Good	C	S

MLS invisible polyinstantiation

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with classified clearance issues a write-up:

```
UPDATE Employee SET Perf = "Great" WHERE Name = "Smith";
```

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Smith	U	40000	C	Great	C	C
Brown	C	80000	S	Good	C	S

Q: Why not just override the original record?

MLS invisible polyinstantiation

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with classified clearance issues a write-up:

```
UPDATE Employee SET Perf = "Great" WHERE Name = "Smith";
```

Name		Salary		Perf		TC
Smith	U	40000	C	Fair	S	S
Smith	U	40000	C	Great	C	C
Brown	C	80000	S	Good	C	S

Q: Why not just override the original record?

A: An explicit approval is needed to merge the instantiations.

MLS visible polyinstantiation

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with secret clearance issues a write-down:

```
UPDATE Employee SET Perf = "Bad" WHERE Name = "Brown";
```

MLS visible polyinstantiation

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with secret clearance issues a write-down:

```
UPDATE Employee SET Perf = "Bad" WHERE Name = "Brown";
```

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S
Brown	C	80000	S	Bad	S	S

MLS visible polyinstantiation

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with secret clearance issues a write-down:

```
UPDATE Employee SET Perf = "Bad" WHERE Name = "Brown";
```

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S
Brown	C	80000	S	Bad	S	S

Q: Why not just override the original record?

MLS visible polyinstantiation

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S

A user with secret clearance issues a write-down:

```
UPDATE Employee SET Perf = "Bad" WHERE Name = "Brown";
```

Name		**Salary**		**Perf**		**TC**
Smith	U	40000	C	Fair	S	S
Brown	C	80000	S	Good	C	S
Brown	C	80000	S	Bad	S	S

Q: Why not just override the original record?

A: An explicit declassification is needed to merge the instantiations.

Outline

1. Background: relational database
2. Access control
3. Integrity
4. Others

Security requirements for a database

- Access control
    - who can read? who can write?
- Authentication
    - how do we know if a DB client is not masquerading as someone else
- Confidentiality
    - what if the DB server is compromised? what about network tapping?
- Integrity
    - how do we guarantee that the data is in an intact and sensible state
- Availability
    - redundancy? fault-tolerance? Byzantine fault tolerance?
- Auditability
    - a.k.a. provenance, proving how we ended up with a specific state

Isn't integrity covered in crypto-protocols?

Isn't integrity covered in crypto-protocols?

We are talking about a different type of integrity here.

- In cryptography: integrity means that data cannot be changed without being detected
- In database: integrity means that the data records are in a sensible/correct state

Isn't integrity covered in crypto-protocols?

We are talking about a different type of integrity here.

- In cryptography: integrity means that data cannot be changed without being detected
- In database: integrity means that the data records are in a sensible/correct state

  We will cover the following types of integrity properties:

  - Element integrity
  - Referential integrity
  - All-or-nothing / Atomicity

Isn't integrity covered in crypto-protocols?

We are talking about a different type of integrity here.

- In cryptography: integrity means that data cannot be changed without being detected
- In database: integrity means that the data records are in a sensible/correct state

  We will cover the following types of integrity properties:
  - Element integrity
  - Referential integrity
  - All-or-nothing / Atomicity

  The goal of ensuring integrity is to prevent users from making changes that will result in an invalid database state. These changes can be either intentional or unintentional.

Element integrity

Example on element integrity violations

```
CREATE TABLE Employee (Name VARCHAR(255), Age INTEGER);
INSERT INTO Employee VALUES ("SMITH", 400);
```

Element integrity

Example on element integrity violations

```
CREATE TABLE Employee (Name VARCHAR(255), Age INTEGER);
INSERT INTO Employee VALUES ("SMITH", 400);
```

Q: What is the problem here? Is it a mistake from developers?

Element integrity

Example on element integrity violations

```
CREATE TABLE Employee (Name VARCHAR(255), Age INTEGER);
INSERT INTO Employee VALUES ("SMITH", 400);
```

Q: What is the problem here? Is it a mistake from developers?

A: The type system is not expressive enough. There is no way to restrict that `Age` must be in a proper range (e.g., 0-150).

Element integrity

Example on element integrity violations

```
CREATE TABLE Employee (Name VARCHAR(255), Age INTEGER);
INSERT INTO Employee VALUES ("SMITH", 400);
```

Q: What is the problem here? Is it a mistake from developers?

A: The type system is not expressive enough. There is no way to restrict that `Age` must be in a proper range (e.g., 0-150).

And there are even more tricky situations, for example:

- At all times, there is at most one employee can have the `Position` attribute set to "CEO".
- A salary increase cannot exceed 100% of the current salary.

Check element integrity with triggers

A typical way to enforce element integrity is to use triggers, i.e., procedures that are automatically executed after each write operation, including `INSERT`, `UPDATE`, `DELETE`, . . . queries

Check element integrity with triggers

A typical way to enforce element integrity is to use triggers, i.e., procedures that are automatically executed after each write operation, including `INSERT`, `UPDATE`, `DELETE`, . . . queries

An example on SQL trigger

```
CREATE TRIGGER AgeCheck ON Employee
    AFTER INSERT, UPDATE
    FOR EACH ROW
    BEGIN
        IF NEW.Age >= 150
        BEGIN
            RAISERROR ("Invalid age")
        END
    END;
```

Foreign key

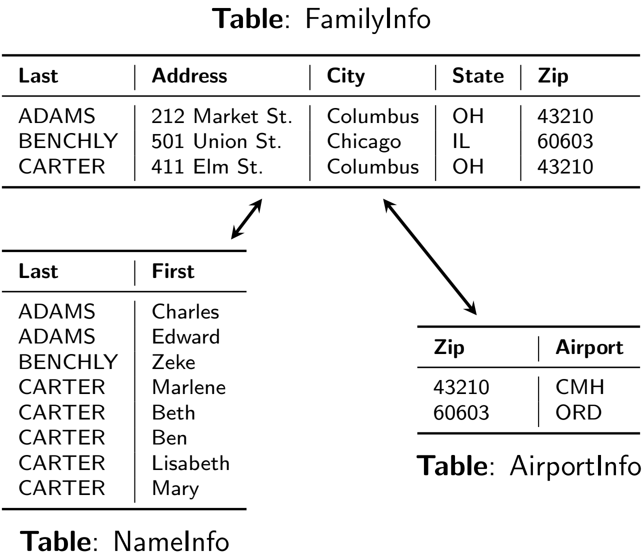


Table: FamilyInfo

Last	**Address**	**City**	**State**	**Zip**
ADAMS	212 Market St.	Columbus	OH	43210
BENCHLY	501 Union St.	Chicago	IL	60603
CARTER	411 Elm St.	Columbus	OH	43210

Last	**First**
ADAMS	Charles
ADAMS	Edward
BENCHLY	Zeke
CARTER	Marlene
CARTER	Beth
CARTER	Ben
CARTER	Lisabeth
CARTER	Mary

Table: NameInfo

Zip	**Airport**
43210	CMH
60603	ORD

Table: AirportInfo

Foreign key

Table: FamilyInfo

Last (PK)	**Address**	**City**	**State**	**Zip (FK)**
ADAMS	212 Market St.	Columbus	OH	43210
BENCHLY	501 Union St.	Chicago	IL	60603
CARTER	411 Elm St.	Columbus	OH	43210

Last (FK)	**First**
ADAMS	Charles
ADAMS	Edward
BENCHLY	Zeke
CARTER	Marlene
CARTER	Beth
CARTER	Ben
CARTER	Lisabeth
CARTER	Mary

Table: NameInfo

Zip (PK)	**Airport**
43210	CMH
60603	ORD

Table: AirportInfo

Foreign key

Foreign key in table creation

```
CREATE TABLE FamilyInfo (
  Last VARCHAR(255) NOT NULL,
  Address VARCHAR(1024),
  City VARCHAR(128),
  State VARCHAR(128),
  Zip VARCHAR(128),
  PRIMARY KEY (Last),
  FOREIGN KEY (Zip) REFERENCES AirportInfo(Zip),
);
```

Foreign key

Foreign key in table creation

```
CREATE TABLE FamilyInfo (
  Last VARCHAR(255) NOT NULL,
  Address VARCHAR(1024),
  City VARCHAR(128),
  State VARCHAR(128),
  Zip VARCHAR(128),
  PRIMARY KEY (Last),
  FOREIGN KEY (Zip) REFERENCES AirportInfo(Zip),
);
```

Q: Why do we need this line here?

Referential integrity

Referential integrity ensures that each value of a foreign key *refers* to a valid primary key value, i.e. there are no dangling foreign keys.

Referential integrity

Referential integrity ensures that each value of a foreign key *refers* to a valid primary key value, i.e. there are no dangling foreign keys.

One use case: to prevent accidental or intentional deletion of records that are still being used.

Referential integrity

Referential integrity ensures that each value of a foreign key *refers* to a valid primary key value, i.e. there are no dangling foreign keys.

One use case: to prevent accidental or intentional deletion of records that are still being used.

Example: dropping a still-in-yuse table

```
DROP TABLE AirportInfo;
```

This operation will raise an error by the DBMS.

Inconsistent state

Recall that integrity is about ensuring the data records are in a sensible/correct state at all times.

But what if a transaction requires two or more write operations? For example: transfer money from Alice to Bob requires two `UPDATE`:

- `UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";`
- `UPDATE Ledger SET Balance = Balance + 100 WHERE Name = "Bob";`

Inconsistent state

Recall that integrity is about ensuring the data records are in a sensible/correct state at all times.

But what if a transaction requires two or more write operations? For example: transfer money from Alice to Bob requires two `UPDATE`:

- `UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";`
- `UPDATE Ledger SET Balance = Balance + 100 WHERE Name = "Bob";`

Q: What happens if the database fails after the first `UPDATE`?

Inconsistent state

Recall that integrity is about ensuring the data records are in a sensible/correct state at all times.

But what if a transaction requires two or more write operations? For example: transfer money from Alice to Bob requires two `UPDATE`:

- `UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";`
- `UPDATE Ledger SET Balance = Balance + 100 WHERE Name = "Bob";`

Q: What happens if the database fails after the first `UPDATE`?

A: The money would be lost forever!

Transaction as an all-or-nothing mechanism

Transaction (abort)

```
BEGIN TRANSACTION;
  UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";
  UPDATE Ledger SET Balance = Balance + 100 WHERE Name = "Bob";
COMMIT TRANSACTION;
```

Transaction as an all-or-nothing mechanism

Transaction (commit or rollback)

```
BEGIN TRANSACTION;
  UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  IF @balance < 100
    BEGIN
      ROLLBACK TRANSACTION;
    END
  ELSE
    BEGIN
      UPDATE Ledger SET Balance = Balance + 100 WHERE Name = "Bob";
      COMMIT TRANSACTION;
    END
```

Data race

Notice that in the prior example, we used an unusual syntax to update the balance:

Atomic update (implicit)

```
UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";
```

Data race

Notice that in the prior example, we used an unusual syntax to update the balance:

Atomic update (implicit)

```
UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";
```

If used on its own (i.e., not in a transaction context), this is implicitly translated into a transaction:

Atomic update (explicit)

```
BEGIN TRANSACTION;
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
```

Data race

Notice that in the prior example, we used an unusual syntax to update the balance:

Atomic update (implicit)

```
UPDATE Ledger SET Balance = Balance - 100 WHERE Name = "Alice";
```

If used on its own (i.e., not in a transaction context), this is implicitly translated into a transaction:

Atomic update (explicit)

```
BEGIN TRANSACTION;
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
```

Q: Why must we enclose it within a transaction?

Data race

If two clients send the request concurrently, what will be the result?

Client 1

```
SELECT @balance = Balance
  FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance =
  @balance - 100 WHERE Name = "Alice";
```

Client 2

```
SELECT @balance = Balance
  FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance =
  @balance - 100 WHERE Name = "Alice";
```

Data race

If two clients send the request concurrently, what will be the result?

Client 1

```
SELECT @balance = Balance
  FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance =
  @balance - 100 WHERE Name = "Alice";
```

Client 2

```
SELECT @balance = Balance
  FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance =
  @balance - 100 WHERE Name = "Alice";
```

One possible interleaving:

Transaction interleavings

```
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
```

Q: How much is deducted from Alice's balance?

Transaction as a serialization mechanism

Transaction interleavings

```
BEGIN TRANSACTION;
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
BEGIN TRANSACTION;
  SELECT @balance = Balance FROM Ledger WHERE Name = "Alice";
  UPDATE Ledger SET Balance = @balance - 100 WHERE Name = "Alice";
COMMIT TRANSACTION;
```

Outline

1. Background: relational database
2. Access control
3. Integrity
4. Others

Security requirements for a database

- Access control
  - who can read? who can write?
- Authentication
  - how do we know if a DB client is not masquerading as someone else
- Confidentiality
  - what if the DB server is compromised? what about network tapping?
- Integrity
  - how do we guarantee that the data is in an intact and sensible state
- Availability
  - redundancy? fault-tolerance? Byzantine fault tolerance?
- Auditability
  - a.k.a. provenance, proving how we ended up with a specific state

Authentication

This is a recap of what we learned from last module. . .

Authentication

This is a recap of what we learned from last module. . .

Q: How does a client authenticate a DBMS server?

Authentication

This is a recap of what we learned from last module. . .

Q: How does a client authenticate a DBMS server?

A: Certificates

Authentication

This is a recap of what we learned from last module. . .

Q: How does a client authenticate a DBMS server?

A: Certificates

Q: How does a DBMS server authenticate a client?

Authentication

This is a recap of what we learned from last module. . .

Q: How does a client authenticate a DBMS server?

A: Certificates

Q: How does a DBMS server authenticate a client?

A: Some possibilities:

- Passwords
- Certificates
- LDAP (Lightweight Directory Access Protocol) server

Confidentiality

Now we have:

- *Authentication*, which reduces the risk that someone gains unauthorized access to the database.
- *Access control*, which further reduces the risks of leakage of secret information.
- *Correctness*, which guarantees that the DBMS software never has a bug (as we see in the Program Security module) and always comply with the policies.

Q: then what else can go wrong?

Confidentiality

The DBMS is simply an application that runs on some OS, along side with other applications.

- Perhaps that machine itself is stolen and an attacker then removes the hard-drive, and attempts to read off the database contents from the hard-drive.

Confidentiality

The DBMS is simply an application that runs on some OS, along side with other applications.

- Perhaps that machine itself is stolen and an attacker then removes the hard-drive, and attempts to read off the database contents from the hard-drive.
- Perhaps that other applications are compromised and attackers simply scan over your file system and extract all files related to the database content.

Confidentiality

The DBMS is simply an application that runs on some OS, along side with other applications.

- Perhaps that machine itself is stolen and an attacker then removes the hard-drive, and attempts to read off the database contents from the hard-drive.
- Perhaps that other applications are compromised and attackers simply scan over your file system and extract all files related to the database content.
- Perhaps that storage provider itself is malicious, especially in the cloud computing setting, and are curious about what you store in your database.

Confidentiality

Solution? If trust is an issue, check if cryptography can be helpful.

Confidentiality

Solution? If trust is an issue, check if cryptography can be helpful.

- File-level encryption
- Column-level encryption

Confidentiality

Solution? If trust is an issue, check if cryptography can be helpful.

- File-level encryption
- Column-level encryption

Q: Obviously the key cannot be stored alongside the data, then in this case, how do you supply the key to the DBMS?

Confidentiality

Solution? If trust is an issue, check if cryptography can be helpful.

- File-level encryption
- Column-level encryption

Q: Obviously the key cannot be stored alongside the data, then in this case, how do you supply the key to the DBMS?

A: Many possible solutions, e.g., establish a secure channel with the DBMS via TLS and send the key, etc.

Availability

Availability is about recognizing the fact that:

- Transactions can fail due to physical problems.
  - System crashes. Disk failures.
  - Physical problems/catastrophes: power failures, floods, fire, thefts.

Availability

Availability is about recognizing the fact that:

- Transactions can fail due to physical problems.
  - System crashes. Disk failures.
  - Physical problems/catastrophes: power failures, floods, fire, thefts.
- Contingency plans are needed to *recover* from these events

High availability in enterprise settings

High availability in enterprise settings

- Redundancy: reduce risk that service is affected from some component failure transparently transfer operations to another functioning component.
    - Uninterrupted power supplies.
    - Multiple hard-drives in RAID configurations (with error-detection codes or error-correction codes).

High availability in enterprise settings

- Redundancy: reduce risk that service is affected from some component failure transparently transfer operations to another functioning component.
  - Uninterrupted power supplies.
  - Multiple hard-drives in RAID configurations (with error-detection codes or error-correction codes).
- Database clusters: Redundancy by more machines.
  Load-balancing among clustered machines.

High availability in enterprise settings

- Redundancy: reduce risk that service is affected from some component failure transparently transfer operations to another functioning component.
  - Uninterrupted power supplies.
  - Multiple hard-drives in RAID configurations (with error-detection codes or error-correction codes).
- Database clusters: Redundancy by more machines.
  Load-balancing among clustered machines.
- Failover: deal with catastrophes etc., when machines are down.
  - Clustered machines are in the same physical location, so all machines may be down.
  - Primary system handles traffic regularly WHILE secondary system takes over in case of failures.

Auditability

Expecting the DBMS will never fail in access control or integrity is a dangerous thought!

Auditability

Expecting the DBMS will never fail in access control or integrity is a dangerous thought!

In the event of a data breach, we want to be able to:

- retroactively identify who has run these queries without authorization.
- hold users accountable and deter such accesses.
- comply with relevant legislation, e.g. HIPAA for health data.

Auditability

- Set an audit policy (or policies) to observe queries received by the DBMS.
- DBMS generates an audit trail or log of events that comply with the audit policy. This log can be processed later into DB tables.
- Archive the audit log periodically to ensure *availability* of the logs for future.