Databases

Hoorcollege 1

INTRODUCTION

A database (DB) is a collection of data with

- A certain logical structure
- A specific semantics
- A specific group of users

A database management system (DBMS) allows to

- Create, modify and manipulate a database
- Query (retrieve) the data using a query language
- Support persistent storage of large amounts of data
- Enable durability and recovery from failure
    - Without unexpected interaction among users(isolation)
    - Actions on the data should never be partial (atomicity)
        - Everybody got something or nobody

Why not just store data in files?

- No query language
- Logical structure is limited to directories
- No efficient access
    - Searching through a large file can take hours
- No or limited protection from data loss
- No access control for parallel manipulation of data
    - (more than one can manipulate the data at the time)

Motivation for dms

- Data independence
    - Logical view on the data independent of physical storage
        - How it is stored on a hard drive, is not important
    - User interacts with a simple view on the data
    - Behind the scenes are complex storage structures that allow rapid access and manipulation
- Avoidance of duplication
    - Different views on the same database
        - For different users or different applications
        - Hiding parts of the data for privacy or security
- High-level declarative query languages

    - Query tells what you want, independent of storage structure
    - Efficient data access (automatic optimisation)

Relational model

Schema: structure of the database = relations + constraints

Example schema:

- Customer(id, name, street, city)
    - Primary key constraint on id
- Account(depositor > customer, accountnr)
    - Foreign key constraint on depositor

Various types of constraints:

- Datatypes, constrained data types, null ability
- Columns constraints
- Check constraints (logical expression for domain integrity)

Instance: actual content (‘state’) of the database at some moment

View of data:

- Different applications might use different views
- Data is stored only once at the physical level
    - Good for consistency

Ansi sparc architecture: 3 levels

- Physical level: how a record is stored
    - Disk pages, index structures, byte layout, record order
- Logical level: also called conceptual schema
    - Describes data stored in the database and
    - Relations among the data
- View level:
    - Application programs hide details of data types
    - Hide information for privacy or security reasons

Data independence

Logical data independence: ability to modify the logical schema without breaking existing applications

- Applications access the views, no the logical database

Physical data independence: ability to modify the physical schema without changing the logical schema:

- E.g. a change in workload might cause the need for
    - Different indexing structures
    - Different database engine

- Distributing the database on multiple machines

Declarative query language:

Queries should:

- Describe what information is sought
- Not prescribe any particular method how to compute/retrieve the desired information

Kowalski: algorithm = logic + control

Imperative/procedural languages:

- Explicit control
- Implicit logic

Declarative/non – procedural languages:

- Implicit control
- Explicit logic
  - E.g Logic programming (prolog), functional programming (Haskell), markup languages (HTML) ….

SQL: structure query language

- Sql is a declarative data manipulation language. The user describes conditions the requested data is required to fulfil.
- Example:
  - SELECT POINTS
  - FROM SOLVED
  - WHERE STUDENT = ‘Ann Smith’
    - AND HOMEWORK = 3
- More concise than imperative languages
  - Less expensive program development
  - Easier maintenance
- Database system will
  - Optimise the query
  - Decides how to execute the query as fast as possible
- (usually) users do not need to think about efficiency

Motivation for DMS:

- Well-defined data models & data integrity constraints
- Entity-relationship models (E/R)
- UML class diagrams
- Relational model
  - E.g. SQL table and constraint definitions
- Meta language for describing
  - Data

    - Data relationships
    - Data semantics
    - Data constraints

Other models:

- Object-oriented models
    - ODL
- Semi-structured data models
    - DTD, XML schema

Entity relationship model

- Entities = objects
    - Customers, accounts, bank branches
- Relationship between entities
    - Account 101343 is held by customer Johnson

UML class diagram

- Frequently used in database design
- Similar to E/R diagrams
    - Instead of entities and relationships in E R model, classes and associations.

Motivation for DMS

- Multiple users, concurrent access
- Transactions with ACID properties

ACID properties:

- Atomicity: transaction executes fully or not at all
- Consistency: database remains in a consistent state where all integrity constraints hold
- Isolation: multiple users can modify the database at the same time but will not see each other’s partial actions

A transaction is a collection of operations that performs a single logical function in a database application

Why DMS?

- Data independence
- Avoidance of duplication
- High-level declarative query languages
- Data models & data integrity
- Persistent storage, safety and high availability
- Scalability
- Security

Relation model

Data values, types and domains

All table entries are data values which conform to some given selection of data types

Examples:

- Strings
- Numbers
- Data and time
- Binary data

The set of available data types depends on:

- DMS
- Supported version of the SQL standard

The Domain val(D) of a type D is the set of possible values

Example:

- Val(numeric(2)) = {-99, … , 99}

In SQL the user may define application-specific domains as names for (subsets of) standard data types:

- Create domain exnum as numeric(2)

We may even add the constraint that the exercise number has to be positive:

- Create domain exnum as numeric(2)
    - Check(value > 0)

Domains are useful to document that two columns represent the same kind of real-world object. So that comparison is meaningful.

Relation schema s (schema of a single relation) defines:

- A (finite) sequence A1, … , An of distinct attribute names
- For each attribute Ai a data type (or domain) Di
    - A relation schema can be written as s = (A1:D1, …, An :Dn)

Let dom(Ai) = val(Di) be the set of possible values for Ai

NOTATION

SQL CREATE TABLE statements represent a rather poor way to communicate schemas (from human to human)

```
CREATE TABLE EXERCISES

(CAT CHAR(1), ENO NUMERIC(2),
TOPIC VARSCHAR(40), MAXPT
NUMERIC(2))
```

If the column data types are not important, don’t write them

A relational database schema S defines:

- A finite set of relation names {R1, … , Rm}
- For every relation Ri, a relation schema sch(Ri)
- A set C of integrity constraints
- S = ({R1, …, Rm}, sch(Ri), C)

Tuples are used to formalize table rows.

A tuple t with respect to the relation schema:

- S =(A1: D1, …, An:Dn)
- Is a sequence (d1, …, dn) of n values such that di ϵ val(Di)

A database state I for this database schema defines for every relation name Ri to a finite set of tuples with respect to the relation schema (Ri)

- You can think of the state as tables conforming to the schema
- Except:
    - There is no order on the tuples
    - Tables contain no duplicate tuples

Database >>> relation/classes/table >>> objects/rows/tuples >>> attribute value/data

Null values

The relational model allows missing attribute values

- Table entries may be empty

NULL = **not** the number 0 or the empty string. A null value is different from all values of any data type.

They are used in a variety of scenarios:

- A value exists but is not kwown
- No value exists
- The attribute is not applicable for this tuple
- Any value will do

Without null values, it would be necessary to split a relation into many, more specific relations (‘subclasses’)

Example:

- Student_with_email / student_without_email

Or introduce an additional relation with schema:

- Stud_email (SID, EMAIL)

This complicates queries. If null values are not allowed, users will invent fake values to fill the missing columns, which is a bad idea because:

- Since the same null value is used for quite different purposes, there can be no clear semantics
- SQL uses a three-valued logic (true, false, unknown) for the evaluation of comparisons that involve null values.
    - For users accustomed to two-valued logic, the outcome is often surprising, common equivalences do not hold.

      SELECT * FROM R WHERE A = 42

      SELECT * FROM R WHERE NOT (A=42)

      Rows with null in column A will not appear in either result.

- Some programming languages do not know about null values. This complicates application programs.
    - When an attribute value is read into a program variable, an explicit null value check and treatment is required
- Since null values may lead to complications, SQL allows to control wether an attribute value may be null or not.
    - By default, null values are allowed
    - Declaring many attributes as not null
        - Leads to simpler application programs
        - Fewer surprises during query evaluation

Constraints: general Remarks

Primary goal of DB design: the database should be an image of the relevant subset of the real world

- The plain definition of tables and columns often allows too many meaningless, illegal database sets

Integrity constraints (IC) are conditions which every database state has to satisfy.

- This restricts the set of possible database states
- Integrity constraints are specified as part of the database schema (component C)

The database management system will refuse any update leading to a database state that violates any of the constraints.

In the SQL create Table statement, the following types of constraints may be specified:

- NOT NULL: No value in this column can be the null value
- KEYS: Each key value can appear once only.
- FOREIGN KEYS: Values in a column must also appear as key values in other tables.
- CHECK: Column values must satisfy a given predicate.
    - SQL allows for inter-column CHECK constraints

Why specify constraints?

- Some protection against data input errors
- Constraints document knowledge about DB states
- Enforcement of law/company standards
- Protection against inconsistency if data is stored redundantly
- Queries/application programs become simpler, if the programmer may assume that the data fulfils certain properties

A **key** of a relation R is an attribute A that uniquely identifies the tuples in R.

The key constraint is satisfied in the DB state I if and only if, for all types $t,u \in I(R)$ the following holds:

- If t. A = u.A then t=u

In other words: different tuples have different values for A

Once a key has been declared for an entity, the DBMS will refuse any insertion of tuples with duplicate key values.

Keys are constraints: they refer to all possible DB states, not only the current one.

In general a key can consist of several attributes; such keys are also called composite keys.

If columns A and B together form a composite key, it is forbidden that there are two tuples $t \neq u$ which agree in both attributes:

- $t.A = u.A \wedge t.B = u.B$

Columns may in agree A or B, though.

Implication between key constraints: a key becomes weaker if attributes are added to the key.

If a set of attributes A satisfies the key constraint, then any superset K that includes A will automatically also have the unique identification property.

A key with attribute set {A1, …, Ak} is minimal if no Ai can be removed from the set without destroying the unique identification property.

The usual definition of keys requires that the set of key attributes {A1, ..., Ak} is minimal.

A relation may have more than one key. A relation may also have more than one minimal key.

In the relational model, one key is designated as the primary key. A primary key cannot be null. All other keys are called alternate or secondary keys.

It is good design practise to define a primary key that consist of a single simple attribute only and will never be updated.

- Good for consistency/applications might store the key

The primary key attributes are often marked by underlining them in the relation schema specifications.

Foreign keys

The relational model does not provide explicit relationships, links or pointers.

Idea: use the key attributes to reference a tuple

- The values for the key attributes uniquely identify a tuple
    - The key attributes values may serve as logical tuple addresses.

To refer to tuples of R in a relation S

- Add the primary key attributes of R to the attributes of S
    - Such reference is only stable if the logical address of a tuple does not change.

A foreign key implements a one-to-many relationship.

In relational databases, a one-to-many relationship occurs when a parent record in one table can potentially reference several child records in another table. In a one-to-many relationship, the parent is not required to have child records; therefore, the one-to-many relationship allows zero child records, a single child record or multiple child records. The important thing is that the child cannot have more than one parent record.

The foreign key constraint ensures the referential integrity of the database.

Once a foreign key is declared, the following update, updates operations violates the foreign key constraint:

- Insertion into table result – without matching tuple in students
    - DBMS rejects the update
- Deletion from table Student – if the deleted tuple is referenced in results
    - DBMS rejects the update, or…
    - Deletion cascades, that is, tuples in results referencing the deleted tuple will also be deleted, or..
    - The foreign key is set to null in results

Only keys may be referenced (primary or secondary)

- References to non-key attributes are not permitted

A table with a composite key must be referenced with a composite foreign key that has the same number of attributes and domains

- It is not required that the corresponding attributes have identical names

Typically, foreign keys are denoted with arrows (→) in relation schema, composite keys appear in parentheses:

- RESULTS(SID → STUDENTS, (CAT, ENO)→ EXERCISES, POINTS)
- Since typically only the primary key is referenced, it is sufficient to simply list the target relation
    - Unless a NOT NULL constraint is present, foreign keys may be null.
        - This corresponds to a nil pointer in programming languages
        - Foreign keys are not themselves keys

Database design

- Formal model of the relevant aspects of the real world
    - Mini world
    - Universe of disclosure
- The real world serves as measure of correctness
    - Possible database states should corresponds to the states of the real world

Database design is challenging:

- Expertise: requires expertise in the application domain
- Flexibility: real world often permits exceptional cases
- Size: database schema may become huge.

Due to complexity, the design is a multi-step process

- Conceptual database design
    - What information do we store
    - How are the information elements related to each other
    - What are the constraints
    - ER/ UML model
- Logical database design
    - Transformation of the conceptual schema into the schema supported by the database
    - Relational model
- Physical database design
    - Design indexes, table distribution, buffer sizes, …
    - To maximise performance of the final system

ER model:

- Entity/objects
    - Is an abstract object:
        - Specific person, company, event
    - rectangles
- Attributes
    - E.G. : people have names and addressess
    - Ellipses
        - Double ellipses
            - represent multi-valued attributes
        - Dashed ellipses
            - Denote derived attributes
- Relationship sets
    - Diamonds
- Lines: link attributes and relationship sets to entity sets
- Underline: indicates primary key attributes

Entity set/class is a collection of similar entities:

- Similar = sharing the same properties/attributes
- E.g: set of all persons, companies, trees, holidays

An entity set is presented by a set of attributes, that is, descriptive properties possessed by all entities of the entity set.

Important difference with object-oriented programming, the E/R model is static, models structure is the structure of data, not the operations and there are no methods/functions associated to entity sets.

Domain: set of permitted values for each attribute

Attribute types:

- Simple and composite attributes:
    - E.g. street is composed of street name and number
- Single-valued and multi-valued attributes
    - Single valued: age of person
    - Multi valued: person can have multiple phone numbers
- Derived attributes
    - Can be computed from other attributes
    - E.g. age, given the birthdate

A relation is an association among several entities

That is, a relationship is a tuple of entities.

A relationship set is a set of relationships of the same kind.

The relationship set connections can be annotated with role indicators, these improve readability.

Cardinality limits express the number of entities to which another entity can be associated via a relationship set.

- Every entity a from A is connected to at least N1, and at most N2 entities in B.
- Every entity b from B is connected to at least M1, and at most M2 entities in A

Cardinality limit	Expression	type	Expression
0 … 1	Zero or one	0 … * R 0 … *	Many to many
1 … 1	Precisely one	0 … 1 R 0 …1	One to one
0 … *	Any number	1…1 R 1…1	One to one
1 … *	At least one	0 … 1 R 0 … *	One to many
		1…1 R 0…*	One to many
		0…* R 0…1	Many to one
		0…* R 1…1	Many to one

Many to Many:

- The entities may be connected arbitrarily
- Every a in A can be linked to an arbitrary number of B’s
- Every b in B can be linked to an arbitrary number of A’s
- If the cardinalities are not given, the default is many-to-many

One to one:

- 0 … 1
  - Every a in A is connected to at most one (=0 or 1) b in B
  - Every b in B is connected to at most one (=0 or 1) a in A
- 1…1
  - Every a in A is connected to precisely one b in B
  - Ever b in B is connected to precisely one a in A

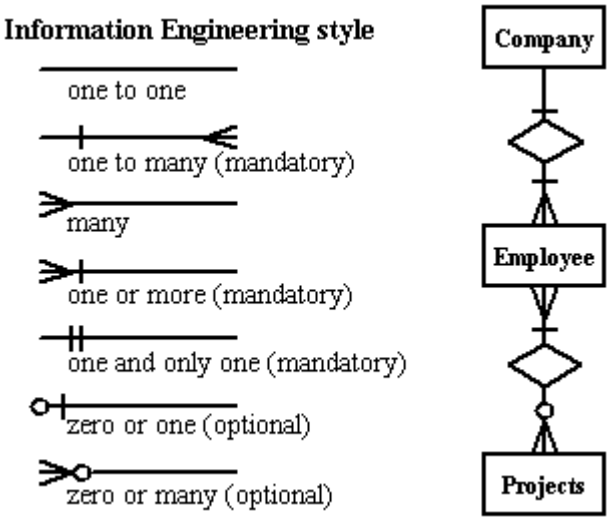
One to many:

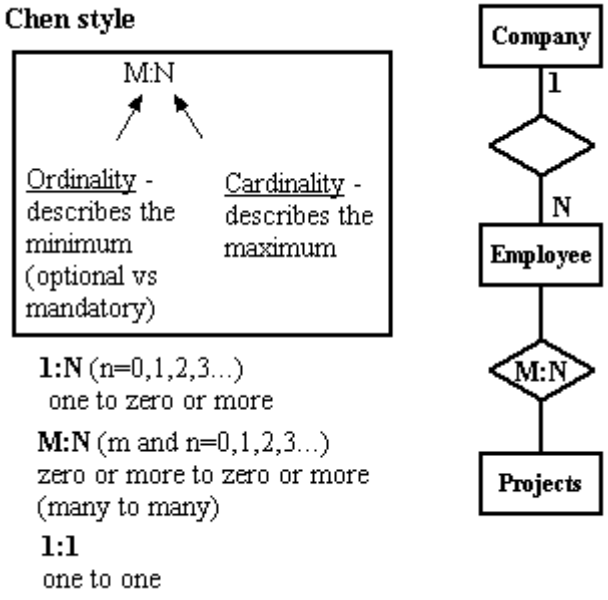
- Every a can have an arbitrary number of links to b in B
- 0 …1
  - Every b in B is connected to at most one a in A
- 1…1

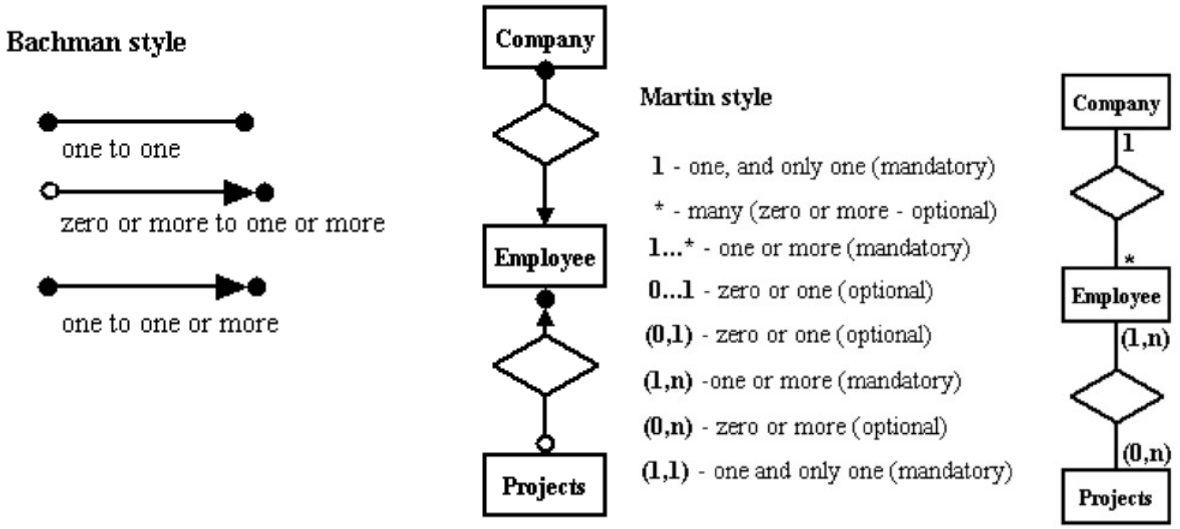
- Every b is connected to precisely one a in A

Many to one

- Every b can have an arbitrary number of links to a in A
- 0…1
  - Every a in A is connected to at most one b in B
- 1…1
  - Every a in A is connected to precisely one b in B







Total participation: means that every entity in the entity set participates in at least on relationship in the relationship set.

Partial participation means that entities may not participate in any relationship in the set.

An attribute can also be property of a relationship set. The value of the relationship attributes is functionally determined by the relationship

Assume that we want to record the date of the last access of a customer to an account. We call this attribute access-date.

- If the relation from customer to account is many-to-many:
  - Access date must be an attribute of depositor
- If the relation from customer to account is one-to-many:
  - Access date can be an attribute of account

The **degree** of a relationship set refers to the number of entity sets participating in the relationship.

- Relationship sets of degree 2 are called binary
- Relationship sets of degree 3 are called ternary

Non-binary relationship sets can be represented using binary ones by creating an artificial entity set.

A **weak entity set** is an entity set without a primary key

- The existence of a weak entity set depends on the existence of an identifying entity set
- There must be a total, one-to-many relationship set from the identifying entity set to the week entity set. This identifying relationship is depicted by a double diamond.
- The discriminator is a partial key, it distinguishes the weak entity only in combination with the identifying entity.

- Primary key of the weak entity set is a combination of the discriminator and primary key of the identifying entity set.

Lower-level entity sets are subgroups of the of higher-level entity sets.Lower-level entity sets inherit all attributes and relationships of the higher-level entity sets

Design principle: specialisation

- Top-down design process
- Identify subgroups within an entity set
- These subgroups become lower-level entity sets which may have attributes or participate in relationships that do not apply to the higher level entity sets

Design principle: generalisation

- Bottom-up design process
- Combine a number of entity sets that share common features into a higher level entity sets

Generalisation and specialisation are both “ is a “ relations

Membership constraints

- Value based: assigns an entity to a specific subclass based on attribute values.
- Default is user-defined: manual assignment to subclasses

Disjointness constraints:

- Disjoint: an entity can belong to at most one subclass
- Default is overlapping: can belong to multiple subclasses

Completeness constraints

- Total specialisation constraint: each superclass entity must belong to a subclass.

Aggregation: treat relationship set as an abstract entry

- Allows relations between relations

UML = unified modelling language

Relational Normal Forms

Functional dependencies (FDs)

- Are a generalization of keys
- Central part of relational database design theory
- This theory defines when a relation is in normal form
    - With respect to a given set of functional dependencies
- It is usually a sign of bad database design if a schema contains relations that violate the normal form
- If a normal form is violated
    - Data is stored redundantly and
    - Information about different concepts is intermixed

Different Normal forms:

- Third Normal Form (3NF) the standard relational normal form used in practice
- Boyce-codd Normal Form (BCNF):
    - A bit more restrictive
    - Easier to define
    - Better for our intuition of good database design
    - Roughly speaking, BNCF requires that all FDs are keys
    - In rare circumstances, a relation might not have an equivalent BCNF form while preserving all its FDs.
        - The 3NF normal form always exists and preserves the FDs

Normalization algorithms can construct good relation schemas from a set of attributes and a set of functional dependencies.

- In practice:
    - Relations are derived from ER models
    - Normalization is used as an additional check only
    - When an ER model is well designed, the resulting derived relational tables will automatically be in BCNF
        - Awareness of normal forms can help to detect design errors already in the conceptual design phase

First Normal Form (1NF): requires that all table entries are atomic – not lists, sets, records, relations.

- The relational model of all table entries are already atomic
- All further normal forms assume that tables are in 1NF

The following are not violations of 1NF:

- A table entry contains values with internal structure
    - Containing a comma separated list
- List represented by several columns
- Nevertheless, these are bad design

FD example: *phone number only depends on the instructor, not on other course data*

INAME {functionally, uniquely} determines PHONE

INAME is a determinant for PHONE

A determinant is a minimal functional dependency.

- Like a partial key:
    - Uniquely determines some attributes, but not all in general

FDS: A1, ..., An >>> B1, ... , Bn

- Sequence of attributes is unimportant
- Both sides formally are sets of attributes
- Holds for a relation R in a database state I if and only if for all tuples t,u ϵ I(R):
    - T.A1 = u.A1 ^ ... ^ t. An = u.An
    - T. B1 = u.B1 ^ ... ^ t.Bm = u.Bm

A key uniquely determines all attributes of its relation.

- There are never two distinct rows with the same key, so the functional dependency condition is trivially satisfied.

Functional dependencies are constraints (like keys). The only functional dependencies are those that hold for all possible states.

- They are generalisation of keys
- Partial keys
- The goal of database normalization is to turn FDs into keys
    - The DBMS is then able to enforce the FDs for the user

During database design, only unquestionable conditions should be used as functional dependencies.

- Database normalization alters the table structure depending on the specified functional dependencies
    - Later hard to change: needs creation/deletion of tables

Implication of functional dependencies

Whenever A → B and B → C, then A → C is automatically satisfied.

FDs of the form A → A always hold

The DB designer is normally not interested in all FDs, but only in a representative FD set that implies all other FDs.

- **Reflexivity:**
  If $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$.
- **Augmentation:**
  If $\alpha \rightarrow \beta$, then $\alpha \cup \gamma \rightarrow \beta \cup \gamma$.
- **Transitivity:**
  If $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.

Simpler way to check wether A→B is implied by an FD set:

- Compute the cover a^+ of a, and
- Then check if b ⊆ a

Cover

The cover of a^+ of a set of attributes a with respect to an FD set F is the set of all attributes B that are uniquely determined by a.

- A set of FDs F implies an FD A → B if and only if B ⊆ A^+

EXAMPLE

Compute the cover {ISBN} + for the following FDs:

- ISBN → Title, Publisher
- ISBN, NO → Author
- Publisher → PUB_URL

1. We start with x = {ISBN}
2. The FD ISBN → Title, Publisher is applicable since, the left-hand side of is completely contained in X.
    a. We get X = {ISBN,TITLE,PUBLISHER}
3. Now the FD Publisher → Pub_url is applicable
    a. We get x = {ISBN, TITLE, PUBLISHER, PUB_URL}
4. No further way to extend set x, the algorithm returns
    a. {ISBN}+ = {ISBN, TITLE, PUBLISHER, PUB_URL}
5. We may now conclude ISBN → PUB_URL

a is a key if the cover a+ contains all attributes. We can use FDs to determine all possible key of R. Normaly we are interested in **minimal** keys only

A key a is minimal if every A ϵ a is vital:

(a-{A})+ ≠*A*

How to determine keys:

RESULTS

STUD_ID	EX_NO	POINTS	MAX_POINTS
100	1	9	10
101	1	8	10
101	2	11	12

$$\mathcal{F} = \left\{ \begin{array}{rcl} \text{STUD\_ID}, \text{EX\_NO} & \rightarrow & \text{POINTS} \\ \text{EX\_NO} & \rightarrow & \text{MAX\_POINTS} \end{array} \right\}$$

We determine a minimal key for the relation RESULTS:

1. $x = \{$ STUD_ID, EX_NO, POINTS, MAX_POINTS $\}$
2. We remove POINTS since $\{$ STUD_ID, EX_NO $\} \subseteq x$:
   $x = \{$ STUD_ID, EX_NO, MAX_POINTS $\}$
3. We remove MAX_POINTS since $\{$ EX_NO $\} \subseteq x$:
   $x = \{$ STUD_ID, EX_NO $\}$
4. Nothing else can be removed. We have a minimal key:
   $\{$ STUD_ID, EX_NO $\}$

Determinants

The attribute set A1, ..., An is called a determinant for attribute set B1, ..., Bm if and only if:

- The FD A1, ..., An → B1, ... , Bm holds, and
- The HIS is minimal, whenevery any Ai is removed then A1, ..., Ai-1, An→B1, ... , Bm does not hold and
- The his and rhs are disntinct

Bad design

- Usually a severe sign of bad DB design if a table contains an FD (encodes a partial function) that is not implied by a key

This lead to:

- Redundant (overtollig) storage of certain facts
  - It wastes storage space
  - Difficult to ensure integrity (betrouwbaarheid) when updating the database

        - All redundant copies need to be updated
        - Wastes time, inefficient
    - Need for additional constraints to guarantee integrity
        - Ensure that the redundant copies indeed agree
    - General FDs are not supported by relational databases
    - The solutions is to transform Fds into key constraints, this is what DB normalization tries to do.
- Insert, update, deletion anomalies (afwijkingen)
    - When a single value needs to be changed multiple tuples must be **updated**. This complicates programs and updates takes longer
    - Redundant copies potentially get out of sync and it is impossible/hard to identify the correct information
    - **Insertion** anomalies arise when unrelated concepts are stored together in a single table
    - When the last course of an instructor **is deleted**, his/her phone number is lost

Boyce-codd Normal Form

A relation R is in Boyce-codd Normal Form (BCNF) if and only if all its FDs are implied by its key constraints.

Alternative characterisation of Boyce-codd Normal form:

BNCF implicates every determinant is a key

- Ensuring its key constraints automatically satisfies all FDs. Hence, no additional constraints are needed
- The anomalies do not occur.

Third Normal Form

A key attribute is an attribute that appears in a minimal key.

- Minimality is important, otherwise all attributes are key attributes

Assume that FDs with multiple attributes on rhs have been expanded. That is, every FD has a single attribute on the right-hand side.

Third Normal form(3NF):

A relation R is in Third normal form (3NF) if and only if every FD A1, … , An → B satisfies at least one of the conditions:

- B ϵ (A1, … , An) the FD is trivial
- {A1, …, An} contains a key of R, or
- B is a key attribute of R

3NF is slightly weaker than BCNF: if a relation is BCNF it is automatically in 3NF

BCNF for every non trivial FD:

- The left-hand side contains a key

3NF for every non trivial FD:

- The left-hand side contains a key, or
- The right –hand side is an attribute of a minimal key
- Every determinant of a non-key attribute is a key

Splitting relations

If a table R is not in BCNF, we can split it into two tables.

- The violating FD determines how to split

If the FD (A1, … , An) → B1, … Bm violates BCNF:

- Create a new relation S(A1,…, An, B1, …,Bm) and
- Remove B1, …, Bm from the original relation R

EXAMPLE

The FD INAME → PHONE is the reason why table

COURSES (CRN, TITLE, INAME, PHONE)

violates BCNF because of INAME → PHONE. We split into:

INSTRUCTORS (CRN, TITLE, INAME)
PHONEBOOK (INAME, PHONE)

It is important that this splitting transformation is lossless, that the original relation can be reconstructed by a join.

When is a split lossless?

Decomposition theorem: the split of relations is guaranteed to be lossless if the intersection of the attributes of the new tables is a key of at least one of them

- The join connects tuples depending on the attribute in the intersection. If these values uniquely identify tuples in the other relation we do not lose information

All splits initiated by the table decomposition method for transforming relations into BCNF satisfy the condition of the decomposition theorem. It is always possible to transform a relation into BCNF by lossless splitting

NOTE: not every lossless split is reasonable

Lossless split guarantees that the resulting schema can represent all DB states that were possible before

- We can translate states from the old into the new schema
- We may stimulate the old schema via views

Lossless splits can lead to more general schemas!

- The new schema allows states which do not corresponds to the state in te old schema

Although **computable columns** lead to violations of BCNF, splitting the relation is not the right solution.

EXAMPLE

Age which is derivable from birthdate, as a consequence we have a functional dependency:

- Birthdate → age

A split would yield a relation:

- R(birthdate, age)

Which would try to materialise the computable function

The correct solution is to eliminate AGE from the table and to define a view which contains all columns plus the computed column age – SQL

Another property for good decomposition of a relation is the **preservation of FDs:**

TRANSACTIONS

ACID properties

Database management systems ensures ACID properties:

- **Atomicity:**
  - Transaction executes fully or not at all
- **Consistency**
  - Transactions always leave the database in a consistent state where all defined integrity constraints hold
- **Isolation**
  - Multiple users can modify the database at the same time but will not see each other's partial actions
- **Durability**
  - Once a transaction is committed successfully, the modified data is persistent, regardless of disk crashes

Anomalies: Lost Update

- The effects of one transaction are lost due to an uncontrolled overwrite performed by a second transaction
- **Inconsistent/dirty read:** a transaction reads the partial result of another transaction

Transaction is a list of actions

The actions are:

- Reads (written R(O))
- Writes (written W(O))
- They end with Commit or Abort

A schedule is a list of actions from a set of transactions

- This is a plan on how to execute transactions

The order in which 2 actions of a transaction T appear in a schedule must be the same order as they appear in T.

A schedule is serial if the actions of the different transactions are not interleaved; they are executed one after another

A schedule is serializable if its effect on the database is the same as that of some serial schedule.

There are several types of conflicts between transactions:

- Write read (WR)
- Read write (RW)
- Write write (WW)

There is a WR conflict between T_1 and T_2 if there is an item Y:

- T_1 write Y and afterwards, T_2 reads Y
- If T_1 has not committed this is a dirty read

There is a RW conflict between T_1 and T_2 if there is an item Y:

- T_1 reads Y and afterwards, T_2 writes Y
- This read becomes unrepeatable

There is a WW between T_1 and T_2 if there is an item Y:

- T_1 writes Y and afterwards, T_2 writes Y
- This write becomes overwritten

Swapping actions

Actions conflict if they:

- Are from different transactions
- Involve the same data item
- One of the actions is a write

We can swap action of different transactions without changing the outcome, if the actions are non-conflicting

Two schedules are conflict equivalent if they can be turned into each other by a sequence of non-conflicting swaps of adjacent actions

Optimising performance

- Rewriting queries
- Change scheduling of queries to reduce contention
- Use a different isolation level for the queries

Read uncommitted

- Only write locks are acquired. Any row read may be concurrently changed by other transactions

Read commited

- Read locks are only held for as long as the application cursor sits on a particular, current row.

Solutions

- Multi-granularity locking
- Declarative locking

Database APIs

Access application via database:

- **Static embedded queries**
  - SQLJ, embedded SQL (C/C++)
  - Pre-processor-based, static SQL
- **Dynamic**
  - JDBC, ODBC, OLE DB, Python DB-API
- **Object relation mapping (ORM)**

- Hide navigational access behind objects
    - JPA/Hibernate, rubyOnrails, ADO.NET/LinQ

Mismatch of types between SQL and a dynamic programming language.

Improving JDBC applications:

- Connection pooling:
    - Keep DB connection open, reduces latency.
- Prepared statements:
    - SQL calls that are repeated often
    - Allows driver to optimise queries
    - Created with Connection.preparestatement()
- Stored procedures to reduce query roundtrips
    - Written in DB-specific language, not portable
    - Accessed with connection.prepareCall()
- Use a driver that is bulk-transfer optimised
    - When retrieving large result sets
    - Driver can send several tuples in a single packet

Dynamic access is vulnerable to SQL injection. Prevented by:

- Never build SQL queries with user input using string concatenation
- Use the API to fill in the query parameters

The **impedance mismatch:** database query language does not match the application programming language.

- In static API
    - Mismatch between SQL and Java types
    - SQL checked for correctness at development time
    - Inflexible
- Dynamic API
    - Mismatch between SQL and Java types
    - Power and flexible but not error-prone
    - SQL query in the string may be incorrect
    - Risk of SQL injection
    - Column names and types unknown at compile time
        - No error checking at development time

Object relational mapping is based on object oriented programming, instead of logical database schemas.

- Table = class
- Row = object
- Foreign key navigation = pointers/references

Mapping from object to database and run-time library handles interaction with the database.

The danger of using ORM is that it can be very inefficient for example if you have to iterate through every object it takes an amount full of time.

To prevent this: HQL queries:

- Allows member access
- They don’t call methods on objects
- Query may return objects

Important aspect of ORM

- Mapping specification:
  - Map relation data onto objects
  - Can to large extends be derived automatically
- Query language
  - Adds object-oriented features to SQL
  - Typically queries as strings
- Persistence:
  - Transaction semantics
  - Languages offer start of transaction, commit and abort
- Fetch strategies
  - Danger of implementing queries in Java
  - Object caching

ORMS are difficult to debug, caused by their complexity. Performance analysis is problematic because the database queries are under the hood and are often very complex.

The ADO.NET entity framework

- Different applications can have different views on the data. These views are entirely implemented on the client side. The advantages are that it avoids polluting DB schema with per-application and there is no added maintenance on the database side.
- Powerful, because a broad set of views are updatable
- Updatability can be statically verified

Uses entity data model (EDM)

- Entity type = structured record with a key
- Entity = instance of an entity type
- Entity types can inherit from other entity types.

After making the model, the EDM is mapped to the logical database schema. This can be queried similar to HQL and JDBC.

LINQ stands for language integrated query. It allows developers to query data structures using an SQL-like syntax. Advantages of LinQ are :

- Queries are first-class citizens and not strings.
- Full type checking and error checking for queries

- LinQ is very readable.
- Allows to query all collection structures
    - DataSet, XML, SQL and Entities

Disadvantage: LinQ is not portable, only available for C# and Visua Basic. This can be helped by similar frameworks in other programming languages.