# Data Modeling and Databases

## Contents

## List of Figures

# 1 Introduction

## 1.1 Motivation

A **database management system** (DMBS) is used to

- avoid redundancy and inconsistency,
- access the data in a declarative manner,
- synchronize concurrent data accesses,
- recover after a system failure,
- guarantee security and privacy and
- facilitate reuse of the data.

## 1.2 Database Abstraction Layers

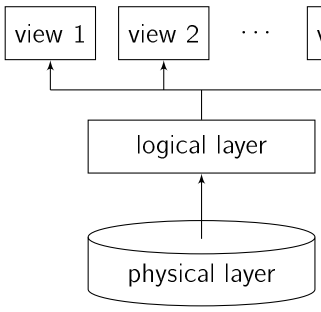

**Figure 1:** The three database abstraction layers.

There are three layers of abstraction:

- The **physical layer** determines how to store data.
- At the **logical layer** the schema of the database determines which data is stored.
- Whereas the schema is an integrated model of the whole information, **views** offer only subsets of this information.

## 1.3 Data Independence

Due to the three layers of abstraction there are two kinds of data independence:

- **Physical data independence** is fuilfilled when modifications of the physical data structures do not affect the logical layer.
- **Logical data independence** is achieved by hiding smaller changes at the logical layer from the views.

Most of the present databases fulfill physical data independence. Logical data independence can only be guaranteed for minor modifications.

## 1.4 Data Modeling

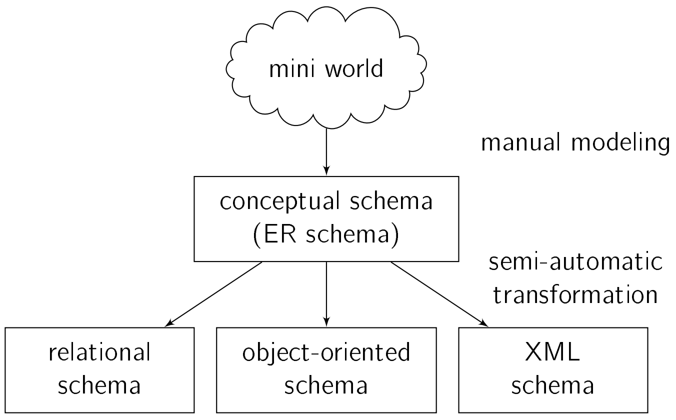

**Figure 2:** Phases of data modeling.

There are three basic categories of models:

- The **conceptual model** is a collection of entities and describes how they relate to each other. It captures the domain to be represented. Commonly used conceptual data models are the entity relationship model and UML.
- The **logical model** (schema) is a mapping of the concepts to a concrete logical representation. Some logical data models are the relational data model, the object-oriented data model and XML.
- The **physical model** is the implementation in a concrete hardware architecture.

# 2 Entity Relationship Model

The **entity relationship model** is a conceptual data model that is used to model the structure of information from the user's point of view.

## 2.1 Building Blocks of an ER Model

### 2.1.1 Entities and Attributes

An **entity** may be defined as an object that is recognized as being capable of an independent existence and can be uniquely identified. Entities can be thought of as nouns.

Similar entities are abstracted as entity types. Entity types are graphically represented by rectangles containing the name of the entity type.

An entity is characterized by one or more **attributes**. Every entity (unless it is a weak entity) must have a minimal set of uniquely identifying attributes, which is called the entity's **key**. Attributes that are part of the key are underlined.

**Example:** In the ER model depicted in figure **??** *student* and *lecture* are entities. *Legi* (key of student), *name*, *semester*, *number* (key of lecture), *title* and *credits* are attributes.

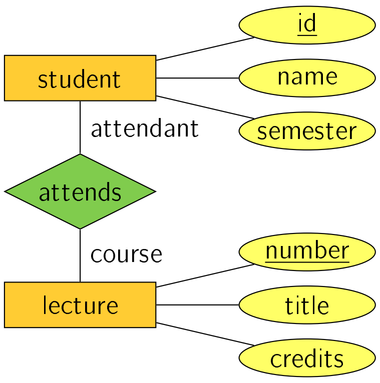

**Figure 3:** Two related entities.

### 2.1.2 Relationships and Roles

A **relationship** captures how entities are related to one another and may be characterized by zero, one or more attributes. Relationships can be thought of as verbs, linking two or more nouns. They are graphically represented by a diamond shape with the name of the relationship inside.

Optionally, **roles** are used to describe how an entity is involved in the relationship.

**Example:** In the ER model depicted in figure **??** *attends* is a relationship and *attendant* and *course* are roles.

### 2.1.3 Weak Entities

An entity that cannot be uniquely identified by its attributes alone is called a **weak entity**. To create a key it must use its attributes in conjunction with the key of an entity it is related to.

**Example:** Since the combination of room number and level is only unique within a building, the building's name is added to the key of room (see figure **??**).

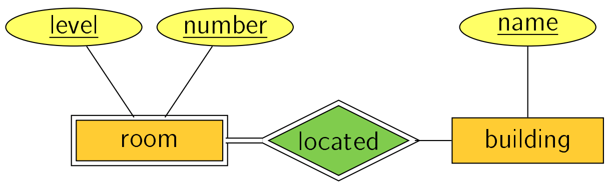

**Figure 4:** An example of a weak entity.

## 2.2 Characterization of Relationships

A relationship $R$ between entities $E_1, E_2, \dots, E_n$ may be considered as a relationship in the mathematical sense:

$$R \subseteq E_1 \times E_2 \times \dots \times E_n$$

The **degree** of a relationship is given by the number of entity types $n$. In practice, most of the relationships are binary.

### 2.2.1 Cardinalities of Binary Relationships

For binary relationships $R \subseteq E_1 \times E_2$ there are three basic **cardinalities**:

- $R$ is a **1:1 relationship** if for every $e_1 \in E_1$ there is at most one $e_2 \in E_2$ such that $(e_1, e_2) \in R$ and vice versa.
- $R$ is a **1:N relationship** if for every $e_2 \in E_2$ there is at most one $e_1 \in E_1$ such that $(e_1, e_2) \in R$. A single element of $E_1$ may be related to multiple elements of $E_2$.
- $R$ is a **N:M relationship** if there are no constraints.

**Note:** Cardinalities are integrity constraints that need hold in the world that is modeled.

### 2.2.2 Cardinalities of $n$-ary Relationships

The concept of cardinalities can be extended to $n$-ary relationships $R \subseteq E_1 \times E_2 \times \dots \times E_n$. A "1" at the entity $E_k$

for some $k \in \{1, \ldots, n\}$ (like in figure **??**) states that the relation $R$ defines a partial function

$$R \quad : \quad E_1 \times \ldots \times E_{k-1} \times E_{k+1} \times \ldots \times E_n \quad \to \quad E_k.$$

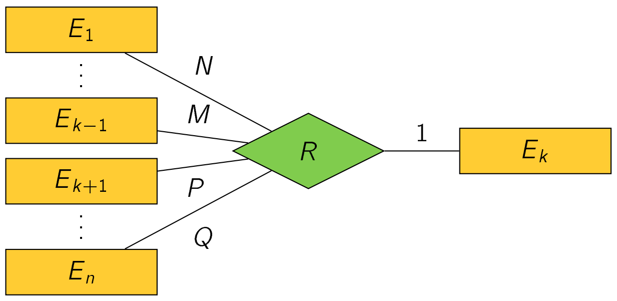

**Figure 5:** Cardinalities of $n$-ary relationships.

### 2.2.3 $(min, max)$-Notation

The **$(min, max)$-notation** is another formalism to characterzie relationships:

For every entity $E_i$ of a relationship $R \subseteq E_1 \times E_2 \times \ldots \times E_n$, a pair $(min_i, max_i)$ of numbers is specified. For all $e_i \in E_i$ at least $min_i$ records $(\ldots, e_i, \ldots)$ exist in $R$ and at most $max_i$ records $(\ldots, e_i, \ldots)$ exist in $R$. Corner cases are:

- If an entity need not participate in the relationship the value of $min$ is set to 0.
- If a single entity may participate arbitrary many times in the relationship then $max$ is replaced with $*$.

## 2.3 Generalization

In a conceptual model **generalization** is used to obtain a more natural and well arranged structure of entities. Properties (attributes and relationships) of similar entity types are "factorized" and assigned to a common supertype. Remaining properties that cannot be "factorized" are assigned to the respective subtype.

A subtype specializes its supertype. A supertype generalizes its subtypes. Entities (instances) of a subtype are implicitly considered as entities of the supertype too.

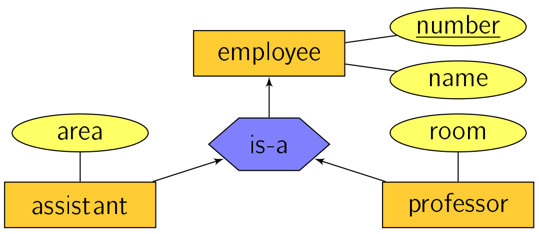

**Figure 6:** Generalization of employees of a university.

## 2.4 Aggregation

TODO

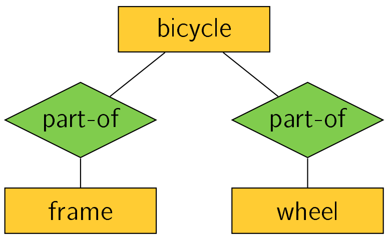

**Figure 7:** Parts of a bicycle.

## 2.5 Consolidation

Often, the world to be modeled is very complex and different views are created by persons who are well versed in a part of this world. These views are generally not disjunct but need to be summarized somehow. This process is called **consolidation**. The resulting schema should be consistent and without redundancy.

# 3 Relational Model

## 3.1 Definition of a Relational Model

### 3.1.1 Mathematical Formalism

Let $D_1, D_2, \ldots, D_n$ be **domains**. These domains may only contain atomic values (e.g. strings and integers, but no sets). A **relation** $R$ is a subset of the Cartesian product of $n$ domains:

$$R \subseteq D_1 \times D_2 \times \ldots \times D_n$$

An element of the set $t \in R$ is called a **tuple**. Since a relation is defined as a set the tuples are unordered and there are no duplicates.

**Note:** The domains do not need to be different, i.e. $D_i = D_j$ is possible for $i \neq j$.

### 3.1.2 Definition of a Schema

A **schema** $\mathcal{R}$ of a relation $R$, sometimes denoted by **sch**$(R)$, is a set of **attributes** $\{A_1, \ldots, A_n\}$ that assigns a name to every component of a tuple $t \in R$.

The names of the attributes have to be unique within a relation. As in the ER model a key is a minimal set of attributes that identify each tuple uniquely.

The domain of the attribute $A_i$ is denoted by **dom**$(A_i)$. Thus, a relation $R$ is a subset of the Cartesian product of the $n$ domains **dom**$(A_1)$, **dom**$(A_2), \ldots,$ **dom**$(A_n)$:

$$R \subseteq \mathbf{dom}(A_1) \times \mathbf{dom}(A_2) \times \ldots \times \mathbf{dom}(A_n)$$

**Example:** An address book can be modeled as a relation

$$\text{addressbook} \subseteq \text{string} \times \text{string} \times \text{integer}.$$

The schema assigns names to the components of a tuple and makes the relation a lot more readable:

$$\text{addressbook} : \{[\text{name} : \text{string}, \text{address} : \text{string}, \underline{\text{tel\#} : \text{integer}}]\}$$

The square brackets $[\ldots]$ are a tuple constructor and the curly brackets $\{\ldots\}$ indicate that the relationship is a set of tuples.

## 3.2 From an ER Model to a Relational Model

### 3.2.1 Relational Representation of Entity Types

Every attribute of the entity becomes an attribute of the relation; keys are directly adopted and stay the same.

**Note:** The set of attributes of a weak entity's relational representation also includes the key of the entity it depends on.

### 3.2.2 Relational Representation of Relationships

The relation $R$ representing the abstract $n$-ary relationship depicted in figure **??** contains all attributes of the relationship and all key attributes of the entities involved.

$$R : \{[\underbrace{A_{1,1}, \ldots, A_{1,k_1}}_{\text{key of } E_1}, \ldots \underbrace{A_{n,1}, \ldots, A_{n,k_n}}_{\text{key of } E_n}, \underbrace{A_1^R, \ldots, A_{k_R}^R}_{\text{attributes of } R}]\}$$

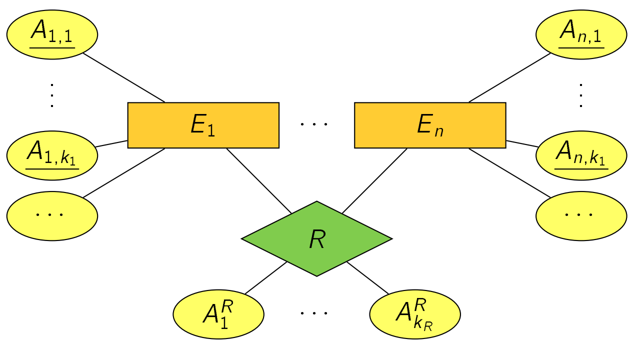

**Figure 8:** Example of a generic $n$-ary relationship.

**Example:** The relationship depicted in figure **??** translates to:

$$\text{attend} : \{[\underline{\text{id} : \text{integer}, \text{number} : \text{integer}}]\}$$

Since this is a N:M relationship *id* as well as *number* are part of the key.

In some situations relations with the same can be summarized. This works relations of 1:1 and 1:N relationships.

## 3.3 Relational Algebra

### 3.3.1 Selection ($\sigma$)

A **selection** is a unary operation $\sigma_{A \circ B}(R)$ or $\sigma_{A \circ v}(R)$, where $R$ is a relation, $A$ and $B$ are attribute names, $v$ is a constant value and $\circ \in \{=, <, \leq, >, \geq, \neq\}$ is a comparison.

The result of the selection $\sigma_{A \circ B}(R)$ contains all tuples in $R$ for which $\circ$ holds between the attributes $A$ and $B$:

$$\sigma_{A \circ B}(R) = \{t \in R \mid t.A \circ t.B\}$$

The result of the selection $\sigma_{A\Phi v}(R)$ contains all tuples in $R$ for which $\Phi$ holds between the attribute $A$ and the constant value $v$:

$$\sigma_{A\Phi V}(R) \quad = \quad \{t \in R \mid t.A\Phi v\}$$

A **generalized selection** is a unary operation $\sigma_\varphi(R)$, where $R$ is a relation and $\varphi$ is a propositional formula that consists of atoms as allowed in the normal selection and the logical operators $\wedge$, $\vee$, and $\neg$. The result of this selection contains all tuples in $R$ for which $\varphi$ holds:

$$\begin{aligned}\sigma_{\varphi_1 \wedge \varphi_2} &= \sigma_{\varphi_1}(R) \cap \sigma_{\varphi_2}(R)\\ \sigma_{\varphi_1 \vee \varphi_2} &= \sigma_{\varphi_1}(R) \cup \sigma_{\varphi_2}(R)\\ \sigma_{\neg\varphi} &= R \setminus \sigma_\varphi(R)\end{aligned}$$

### 3.3.2 Projection ($\pi$)

The **projection** operator $\pi_{A_1,\ldots,A_n}$ is defined as

$$\pi_{A_1,\ldots,A_n}(R) \quad = \quad \{t[A_1,\ldots,A_n] \mid t \in R\},$$

where $t[A_1,\ldots,A_n]$ is the tuple $t$ restricted to the attributes $A_1,\ldots,A_n$.

### 3.3.3 Cartesian Product ($\times$)

The **Cartesian product** $R \times S$ contains all $|R| \cdot |S|$ possible combinations of tuples in the relations $R$ and $S$:

$$R \times S \quad = \quad \{[r_1,\ldots,r_n,s_1,\ldots,s_m] \mid [r_1,\ldots,r_n] \in R, [s_1,\ldots,s_m] \in S\}$$

**Note:** The Cartesian product is defined differently from the one in set theory. The Cartesian product of an $n$-tuple by an $m$-tuple is flattened into an $(n+m)$-tuple.

The schema of the Cartesian product $R \times S$ is

$$\textbf{sch}(R \times S) \quad = \quad \textbf{sch}(R) \cup \textbf{sch}(S) \quad = \quad \mathcal{R} \cup \mathcal{S}.$$

If the attribute $A$ appears in $\mathcal{R}$ as well as in $\mathcal{S}$, they are renamed as $R.A$ and $R.B$; otherwise their names would not be unique in $\mathcal{R} \cup \mathcal{S}$.

### 3.3.4 Rename Operator ($\rho$)

The operator $\rho_{R'}(R)$ **renames** the relation $R$ as $R'$. And the operator $\rho_{A' \leftarrow A}$ that renames a single attribute is defined as

$$\rho_{A' \leftarrow A}(R) \quad = \quad \{t[A' \leftarrow A] \mid t \in R\},$$

where $t[A' \leftarrow A]$ is the tuple $t$ with the attribute $A$ renamed as $A'$.

**Example:** In some cases renaming is necessary. The following combination of operators returns the name of all persons and the name the persons' fathers:

$$\pi_{P_1.\text{name},P_2.\text{name}}(\sigma_{P_1.\text{father}=P_2.\text{id}}(\rho_{P_1}(\text{person}) \times \rho_{P_2}(\text{person})))$$

### 3.3.5 Set Operators ($\cup$, $\cap$ and $\setminus$)

Let $R$ and $S$ be two relations with the same schema (i.e. $\mathcal{R} = \mathcal{S}$). Just like in set theory, the **union**, **intersection** and **difference** operators are

$$\begin{aligned}R \cup S &= \{t \mid t \in R \vee t \in S\}\\ R \cap S &= \{t \mid t \in R \wedge t \in S\}\\ R \setminus S &= \{t \mid t \in R \wedge t \notin S\}.\end{aligned}$$

### 3.3.6 Join Operators

There is a variety of different **join** operators:

- The **theta join** $R \bowtie_\vartheta S$, where $\vartheta$ is an arbitrary join predicate, can be defined as

$$R \bowtie_\vartheta S \quad = \quad \sigma_\vartheta(R \times S).$$

- The **natural join** of two relations $R$ and $S$ is denoted by $R \bowtie S$. Without loss of generality $R$ is assumed to have $n+k$ attributes $A_1,\ldots,A_n,B_1,\ldots,B_k$ and $S$ the $m+k$ attributes $B_1,\ldots,B_k,C_1,\ldots,C_m$ (where all $A_i$ and $C_j$ are pairwise different). The natural join can be defined as

$$R \bowtie S \quad = \quad \pi_{\mathcal{R}\cup\mathcal{S}}(\sigma_\vartheta(R \times S)),$$

where $\vartheta \equiv \bigwedge_{i=1}^{k} R.B_i = S.B_i$.

$$\begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{R}\\ \hline A & B\\ \hline a_1 & b_1\\ a_2 & b_2\\ \hline\end{array} \bowtie \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{S}\\ \hline B & C\\ \hline b_1 & c_1\\ b_3 & c_2\\ \hline\end{array} = \begin{array}{|c|c|c|}\hline \multicolumn{3}{|c|}{R \bowtie S}\\ \hline A & B & C\\ \hline a_1 & b_1 & c_1\\ \hline\end{array}$$

- The **left outer join** $R ⟕ S$ is like the natural join, but all tuples of the left argument's relation $R$ are preserved.

$$\begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{R}\\ \hline A & B\\ \hline a_1 & b_1\\ a_2 & b_2\\ \hline\end{array} ⟕ \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{S}\\ \hline B & C\\ \hline b_1 & c_1\\ b_3 & c_2\\ \hline\end{array} = \begin{array}{|c|c|c|}\hline \multicolumn{3}{|c|}{R ⟕ S}\\ \hline A & B & C\\ \hline a_1 & b_1 & c_1\\ a_2 & b_2 & -\\ \hline\end{array}$$

- The **right outer join** $R ⟖ S$ is analogous to the left outer join.

$$\begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{R}\\ \hline A & B\\ \hline a_1 & b_1\\ a_2 & b_2\\ \hline\end{array} ⟖ \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{S}\\ \hline B & C\\ \hline b_1 & c_1\\ b_3 & c_2\\ \hline\end{array} = \begin{array}{|c|c|c|}\hline \multicolumn{3}{|c|}{R ⟖ S}\\ \hline A & B & C\\ \hline a_1 & b_1 & c_1\\ - & b_3 & c_2\\ \hline\end{array}$$

- The **outer join** ⟗ can be defined as

$$R ⟗ S \quad = \quad (R ⟕ S) \cup (R ⟖ S).$$

In this sense the outer join combines the left and right outer join.

| R | |
|---|---|
| A | B |
| $a_1$ | $b_1$ |
| $a_2$ | $b_2$ |

⟗

| S | |
|---|---|
| B | C |
| $b_1$ | $c_1$ |
| $b_3$ | $c_2$ |

=

| $R ⟗ S$ | | |
|---|---|---|
| A | B | C |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | – |
| – | $b_3$ | $c_2$ |

- The **left semi join**

$$R \ltimes S \quad = \quad \pi_{\mathcal{R}}(R \bowtie S)$$

is the natural join of $R$ and $S$ projected onto the attributes of $R$.

| R | |
|---|---|
| A | B |
| $a_1$ | $b_1$ |
| $a_2$ | $b_2$ |

$\ltimes$

| S | |
|---|---|
| B | C |
| $b_1$ | $c_1$ |
| $b_3$ | $c_2$ |

=

| $R \ltimes S$ | |
|---|---|
| A | B |
| $a_1$ | $b_1$ |

- The **right semi join** $R \rtimes S$ is analogous to the left semi join.

| R | |
|---|---|
| A | B |
| $a_1$ | $b_1$ |
| $a_2$ | $b_2$ |

$\rtimes$

| S | |
|---|---|
| B | C |
| $b_1$ | $c_1$ |
| $b_3$ | $c_2$ |

=

| $R \rtimes S$ | |
|---|---|
| B | C |
| $b_1$ | $c_1$ |

### 3.3.7 Relational Division ($\div$)

Let $R$ and $S$ be two relations such tat $\mathcal{S} \subseteq \mathcal{R}$. The result of the **relational division** $R \div S$ consists of the restrictions of tuples in $R$ to $\mathcal{R} \setminus \mathcal{S}$, for which it holds that all their combinations with tuples in $S$ are represented in $R$.

| R | |
|---|---|
| A | B |
| $a_1$ | $b_1$ |
| $a_1$ | $b_2$ |
| $a_1$ | $b_3$ |
| $a_2$ | $b_2$ |
| $a_2$ | $b_3$ |

$\div$

| S |
|---|
| B |
| $b_1$ |
| $b_2$ |

=

| $R \div S$ |
|---|
| A |
| $a_1$ |

The relational division can be simulated with basic operators as follows:

$$R \div S \quad = \quad \pi_{\mathcal{R}\setminus\mathcal{S}}(R) \setminus \pi_{\mathcal{R}\setminus\mathcal{S}}((\pi_{\mathcal{R}\setminus\mathcal{S}}(R) \times S) \setminus R)$$

## 3.4 Relational Calculus

The **relational calculus** is based on the first-order predicate calculus. There are two equally powerful relational calculi:

- tuple relational calculus and
- domain relational calculus.

### 3.4.1 Tuple Relational Calculus

Expressions in **tuple relational calculus** are of the form

$$\{t \mid P(t)\} \quad \text{or} \quad \{[t_1.A_1, \dots, t_n.A_n \mid P(t_1, \dots, t_n)\},$$

where $t, t_1, \dots, t_n$ are tuple variable (note that $t_i = t_j$ is possible for $i \neq j$), $A_1, \dots, A_n$ are attribute names and $P(t)$ is a predicate formula. Atoms of such a formula are

- $u \in R$, where $u$ is a tuple variable and $R$ is a name of a relation and
- $u.A \circ v.B$ or $u.A \circ c$, where $u$ and $v$ are tuple variables, $A$ and $B$ attribute names, $\circ \in \{=, <, \leq, >, \geq, \neq\}$ a comparison and $c$ is a constant.

Formulae are constructed by the following rules:

- All atoms are formulae.
- If $P$ is a formula, then $\neg P$ and $(P)$ are formulae too.
- If $P_1$ and $P_2$ are formulae, then $P_1 \wedge P_2$, $P_1 \vee P_2$ and $P_1 \Rightarrow P_2$ are formulae too.
- If $P(t)$ is a formula with a free variable $t$, then

$$\forall t \in R(P(t)) \qquad \text{and} \qquad \exists t \in R(P(t))$$

are formulae too; note that the tuple variable $t$ of the quantifications is bound to the relation $R$.

**Example:** The expression

$$\{s \mid s \in \text{student} \wedge \neg(\exists a \in \text{attend}(s.\text{id} = a.\text{id}))\}$$

lists all students that are attending no lectures.

### 3.4.2 Safe Expressions in Tuple Relational Calculus

Some expressions in tuple relational calculus yield infinite results. An example of such an expression is

$$\{t \mid \neg(t \in R)\}.$$

This is not desirable. An expression is considered **safe** if its result is a subset of the domain of the formula. The domain of a formula is the set of all constants and the relations' values that appear in the formula. The result of a safe expression is always finite.

### 3.4.3 Domain Relational Calculus

Expressions of **domain relational calculus** are of the form

$$\{[v_1, \dots, v_n] \mid P(v_1, \dots, v_n)\},$$

where each $v_i$ (for $1 \leq i \leq n$) is a domain variable and $P(v_1, \dots, v_n)$ denotes a predicate over the free variables $v_1, \dots, v_n$.

Atoms of formulae in domain relational calculus are

- $[w_1, \ldots, w_m] \in R$, where $R$ is an $m$-ary relation and
- $x \circ y$ and $x \circ c$, where $x$ and $y$ are domain variables, $c$ is a constant and $\circ \in \{=, <, \leq, >, \geq, \neq\}$ is a comparison.

Formulae are constructed according to the following rules:

- An atom is a formula.
- If $P$ is a formula, then $\neg P$ and $(P)$ are formulae too.
- If $P_1$ and $P_2$ are formulae, then $P_1 \wedge P_2$, $P_1 \vee P_2$ and $P_1 \Rightarrow P_2$ are formulae too.
- If $P(v)$ is a formula with a free variable $t$, then
$$\forall v(P(v)) \qquad \text{and} \qquad \exists v(P(v))$$
are formulae too.

**Example:** The expression

$$\begin{aligned}\{[i, n] \quad &| \quad \exists s([i, n, s] \in \text{student} \\ &\wedge \quad \exists n'([i, n'] \in \text{attends} \\ &\wedge \quad \exists t, c([n', t, c] \in \text{lecture} \\ &\wedge \quad c \geq 8)))\}\end{aligned}$$

lists the id and name of all students that attend a lecture with more at least 8 credits (compare with ER model in figure **??**).

### 3.4.4 Safe Expressions in Domain Relational Calculus

Like in the tuple relational calculus the domain relational calculus allows infinite results:

$$\{[v_1, \ldots, v_n] \mid \neg([v_1, \ldots, v_n] \in R)\}$$

An expression $\{[v_1, \ldots, v_n \mid P(v_1, \ldots, v_n)\}$ is **safe**, if the following conditions hold:

- If the tuple $[v_1, \ldots, v_{i-1}, c_i, v_{i+1}, \ldots, v_n]$, where $c_i$ is a constant, is contained in the result, then $c_i$ must be contained in the domain of $P$.
- Every existentially quantified subformula $\exists v(P'(v))$ may only accept elements of the domain of $P'$. In other words: If $P'(c)$ holds for a constant value, then $c$ is contained in the domain of $P'$.
- Every universally quantified subformula $\forall v(P'(v))$ holds, if and only if $P'(v)$ holds for every value in the domain of $P'$. In other words: $P'(c)$ holds for all values $c$ that are not contained in the domain of $P'$.

## 3.5 Codd's Theorem

**Codd's theorem** states that

- relational algebra,
- tuple relational calculus restricted to safe expressions and
- domain relational calculus restricted to safe expressions

are precisely equivalent in expressive power. That is, a database query can be formulated in one language if and only if it can be expressed in the others.

Query languages that are equivalent in expressive power to relational algebra are called **relationally complete**.

# 4 Relational Design Theory

## 4.1 Functional Dependencies

A **functional dependency (FD)** is a constraint between two sets of attributes in a relation of a database.

Let $\mathcal{R}$ be the schema of a relation $R$. A set of attributes $\alpha \subseteq \mathcal{R}$ is said to functionally determine another set of attributes $\beta \subseteq \mathcal{R}$, if and only if

$$\forall r, t \in R : \quad r.\alpha = t.\alpha \quad \Rightarrow \quad r.\beta = t.\beta.$$

Such a functional dependency is denoted by

$$\alpha \to \beta$$

**Note:** $XY \to Z$ is a sloppy notation for $\{X, Y\} \to \{Z\}$.

A functional dependency $\alpha \to \beta$ is **minimal**, written $\alpha \overset{\bullet}{\to} \beta$, if and only if

$$\forall A \in \alpha : \quad \alpha \setminus \{A\} \not\to \beta.$$

A functional dependency $\alpha \to \beta$ is **trivial**, if $\beta \subseteq \alpha$.

### 4.1.1 Armstrong's Axioms and Conclusions

The three **Armstrong's axioms** are used to infer all the functional dependencies:

- **Reflexivity:** $\beta \subseteq \alpha \Rightarrow \alpha \to \beta$
- **Augmentation:** $\alpha \to \beta \Rightarrow \alpha\gamma \to \beta\gamma$
- **Transitivity:** $\alpha \to \beta \wedge \beta \to \gamma \Rightarrow \alpha \to \gamma$

The Armstrong axioms are sound and complete. Some other rules that can be concluded from these axioms are:

- **Union rule:** $\alpha \to \beta \wedge \alpha \to \gamma \Rightarrow \alpha \to \beta\gamma$
- **Decomposition:** $\alpha \to \beta\gamma \Rightarrow \alpha \to \beta \wedge \alpha \to \gamma$
- **Pseudo transitivity:** $\alpha \to \beta \wedge \beta\gamma \to \delta \Rightarrow \alpha\gamma \to \delta$

**Note:** The closure $F^+$ of a set of functional dependencies $F$ contains all functional dependencies that can be inferred from $F$, using the rules described above.

### 4.1.2 Closure of Attributes

The **closure** of a set of attributes $\alpha$, denoted by $\alpha^+$, is the largest set of attributes that is functionally determined by $\alpha$.

The following algorithm takes a set of functional dependencies $F$ and a set of attributes $\alpha$ as input and computes the closure $\alpha^+$ such that $\alpha \to \alpha^+$:

**AttrClosure($F$,$\alpha$)**

```
$\alpha^+ := \alpha$
while $\alpha^+$ has changed do
  for all $\beta \to \gamma \in F$ do
    if $\beta \subseteq \alpha^+$ then
      $\alpha^+ := \alpha^+ \cup \gamma$
    end if
  end for
end while
return $\alpha^+$
```

### 4.1.3 Minimal Basis

Two set $F$ and $G$ of functional dependencies are equivalent, denoted by $F \equiv G$, if and only if their closures are the same, i.e. $F^+ = G^+$.

$F_c$ is called **minimal basis** of $F$ if and only if:

1. $F_c \equiv F$, i.e. the closure of $F_c$ and $F$ are the same.
2. There is no functional dependency $\alpha \to \beta \in F_c$ that contains unnecessary attributes:
   (a) $\forall A \in \alpha : F_c \not\equiv F_c \setminus (\alpha \to \beta) \cup ((\alpha \setminus \{A\}) \to \beta)$
   (b) $\forall B \in \beta : F_c \not\equiv F_c \setminus (\alpha \to \beta) \cup (\alpha \to (\beta \setminus \{B\}))$
3. There are no two functional dependencies in $F_c$ with the same left side. This can be achieved by applying the union rule.

Given a set of functional dependencies $F$ the minimal basis $F_c$ of $F$ is computed as follows:

1. For every $\alpha \to \beta \in F$ perform a reduction of the left side; i.e. if for an $A \in \alpha$ the condition
   $$\beta \subseteq \text{AttrClosure}(F, \alpha \setminus \{A\})$$
   holds, then replace $\alpha \to \beta$ with $(\alpha \setminus \{A\}) \to \beta$.
2. For every remaining $\alpha \to \beta \in F$ perform a reduction of the right side; i.e. if for a $B \in \beta$ the condition
   $$B \in \text{AttrClosure}(F \setminus (\alpha \to \beta) \cup (\alpha \to (\beta \setminus \{B\})), \alpha)$$
   holds, then replace $\alpha \to \beta$ with $\alpha \to (\beta \setminus \{B\})$.
3. Remove all functional dependencies of the form $\alpha \to \emptyset$.
4. If there are functional dependencies with the same right side $\alpha \to \beta_1, \ldots, \alpha \to \beta_n$, apply the union rule to replace them with a single functional dependency $\alpha \to \cup_{i=1}^{n} \beta_i$.

## 4.2 Multivalued Dependencies

Let $\mathcal{R}$ be the schema of a relation $R$. Furthermore let $\alpha \subseteq \mathcal{R}$ and $\beta \subseteq \mathcal{R}$ be two subsets of $\mathcal{R}$ and $\gamma = \mathcal{R} \setminus (\alpha \cup \beta)$. The **multivalued dependency (MVD)**

$$\alpha \twoheadrightarrow \beta$$

holds, if and only if for all pairs of tuples $t_1$ and $t_2$ in $R$ with $t_1.\alpha = t_2.\alpha$, there exist tuples $t_3$ and $t_4$ in $R$ such that

$$\begin{aligned} t_1.\alpha = t_2.\alpha &= t_3.\alpha = t_4.\alpha \\ t_3.\beta &= t_2.\beta \\ t_3.\gamma &= t_1.\gamma \\ t_4.\beta &= t_1.\beta \\ t_4.\gamma &= t_2.\gamma. \end{aligned}$$

In other words: If there are two tuples with the same $\alpha$-values, their $\beta$-values can be swapped and the resulting tuples have to exist in the relation.

**Example:** In the following relation $R$, the multivalued dependencies $A \twoheadrightarrow B$ and $A \twoheadrightarrow C$ hold:

| $R$ | | |
|---|---|---|
| $A$ | $B$ | $C$ |
| $a_1$ | $b_1$ | $c_1$ |
| $a_1$ | $b_2$ | $c_2$ |
| $a_1$ | $b_2$ | $c_1$ |
| $a_1$ | $b_1$ | $c_2$ |

A multivalued dependency $\alpha \twoheadrightarrow \beta$ is **trivial**, if and only if $\beta \subseteq \alpha$ or $\beta = \mathcal{R} \setminus \alpha$.

### 4.2.1 Properties of Multivalued Dependencies

Some rules for multivalued dependencies are:

- **Complement:** $\alpha \twoheadrightarrow \beta \Rightarrow \alpha \twoheadrightarrow \mathcal{R} \setminus (\alpha \cup \beta)$
- **M.v. augmentation:** $\alpha \twoheadrightarrow \beta \wedge \delta \subseteq \gamma \Rightarrow \alpha\gamma \twoheadrightarrow \beta\delta$
- **M.v. transitivity:** $\alpha \twoheadrightarrow \beta \wedge \beta \twoheadrightarrow \gamma \Rightarrow \alpha \twoheadrightarrow \gamma \setminus \beta$
- **Generalization:** $\alpha \rightarrow \beta \Rightarrow \alpha \twoheadrightarrow \beta$
- **Coalesce:** $\alpha \twoheadrightarrow \beta \wedge \gamma \subseteq \beta \wedge \delta \cup \beta = \emptyset \wedge \delta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$
- **M.v. union:** $\alpha \twoheadrightarrow \beta \wedge \alpha \twoheadrightarrow \gamma \Rightarrow \alpha \twoheadrightarrow \beta\gamma$
- **Intersection:** $\alpha \twoheadrightarrow \beta \wedge \alpha \twoheadrightarrow \gamma \Rightarrow \alpha \twoheadrightarrow \beta \cap \gamma$
- **Difference:** $\alpha \twoheadrightarrow \beta \wedge \alpha \twoheadrightarrow \gamma \Rightarrow \alpha \twoheadrightarrow \beta \setminus \gamma$

**Note:** The closure $D^+$ of a set of functional and multivalues dependencies $D$ contains all dependencies that can be derived from $D$, using the rules described above and in section **??**.

## 4.3 Keys

Let $\mathcal{R}$ be the schema of a relation $R$. A set of attributes $\alpha \subseteq \mathcal{R}$ is a **superkey**, if and only if

$$\alpha \rightarrow \mathcal{R}.$$

I.e. determines all other attributes in $\mathcal{R}$. If $\alpha \overset{\bullet}{\rightarrow} \mathcal{R}$ holds, then $\alpha$ is called a **candidate key** of $\mathcal{R}$.

**Note:** In order to check whether $\kappa$ is a superkey of $\mathcal{R}$ with respect to the functional dependencies $F$, the closure $\kappa^+ = \text{AttrClosure}(F, \kappa)$ is determined. $\kappa$ is a superkey if $\kappa^+ = \mathcal{R}$.

## 4.4 Bad Relational Schemas

Badly designed schemas may lead to **anomalies**. There are three kinds of anomalies:

- **Update anomaly:** Redundancy may lead to inconsistency when data is updated. This is also a place complexity and performance issue.
- **Insert anomaly:** TODO
- **Delete anomaly:** TODO

## 4.5 Decomposition of Relations

A set of schemas $\{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n\}$ is a **decomposition** of a schema $\mathcal{R}$, if all attributes are preserved, i.e.

$$\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \ldots \cup \mathcal{R}_n.$$

The corresponding decomposition of a relation $R$ into the set of relations $\{R_1, R_2, \dots, R_n\}$ is defined as

$$R_i := \pi_{\mathcal{R}_i}(R) \qquad \text{for } 1 \leq i \leq n.$$

### 4.5.1 Lossless Decompositions

A decomposition of $\mathcal{R}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$ is **lossless**, if

$$R = R_1 \bowtie R_2$$

holds for every instance $R$ of $\mathcal{R}$.

A decomposition of $\mathcal{R}$ with functional dependencies $F_{\mathcal{R}}$ into $\mathcal{R}_1$ and $\mathcal{R}_2$ is lossless, if it is possible to infer

$$(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_1 \in F_{\mathcal{R}}^+ \quad \text{or} \quad (\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_2 \in F_{\mathcal{R}}^+.$$

**Note:** This is a sufficient, but not a necessary condition.

### 4.5.2 Preservation of Dependencies

The decomposition $\{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n\}$ of $\mathcal{R}$ with functional dependencies $F_{\mathcal{R}}$ is **dependency preserving** if

$$F_{\mathcal{R}} \equiv (F_{\mathcal{R}_1} \cup F_{\mathcal{R}_2} \cup \ldots \cup F_{\mathcal{R}_n}),$$

where $F_{\mathcal{R}_i} = \{\alpha \rightarrow \beta \in F_{\mathcal{R}}^+ \mid \alpha \cup \beta \subseteq \mathcal{R}_i\}$ for $1 \leq i \leq n$.

TODO

## 4.6 Normal Forms

### 4.6.1 First Normal Form

The **first normal form (1NF)** requires that domains contain only atomic values. This holds due to the definition of the relational model.

### 4.6.2 Second Normal Form

Let $\kappa_1, \kappa_2, \ldots, \kappa_n$ be the candidate keys of the schema $\mathcal{R}$ with functional dependencies $F$. The schema $\mathcal{R}$ is in **second normal form (2NF)**, if and only if

$$\kappa_i \overset{\bullet}{\to} A \in F^+$$

holds for every $1 \leq i \leq n$ and $A \in \mathcal{R} \setminus \{\kappa_1 \cup \kappa_2 \cup \ldots \cup \kappa_n)$. I.e. every non key attribute is minimally dependent on every key.

**Note:** 2NF implies 1NF

### 4.6.3 Third Normal Form

A schema $\mathcal{R}$ is in **third normal form (3NF)**, if for every functional dependency of the form $\alpha \to A$, with $\alpha \subseteq \mathcal{R}$ and $A \in \mathcal{R}$, at least on of the following conditions holds:

- $A \in \alpha$, i.e. the functional dependency is trivial.
- The attribute $A$ is contained in a candidate key of $\mathcal{R}$.
- $\alpha$ is a superkey of $\mathcal{R}$.

**Note:** 3NF implies 2NF.

The following **synthesis algorithm** takes a schema $\mathcal{R}$ and a set of functional dependencies $F$ as input and computes a lossless and dependency preserving decomposition $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ of $\mathcal{R}$, such that every $\mathcal{R}_i$ is in 3NF:

1. Compute the minimal basis $F_c$ of $F$.
2. For every $\alpha \to \beta \in F_c$:
   (a) Create a relational schema $\mathcal{R}_\alpha := \alpha \cup \beta$
   (b) Define $F_\alpha := \{\alpha' \to \beta' \in F_c \mid \alpha' \cup \beta' \subseteq \mathcal{R}_\alpha\}$ to be the functional dependencies of $\mathcal{R}_\alpha$.
3. If none of the schemas $\mathcal{R}_\alpha$ contains a candidate key of $\mathcal{R}$ with respect to $F_c$: Choose a candidate key $\kappa \subseteq \mathcal{R}$ and define the schema $\mathcal{R}_\kappa$ with $F_\kappa := \emptyset$.
4. Eliminate all schemas $\mathcal{R}_\alpha$ that are contained in another schema $\mathcal{R}_{\alpha'}$ (i.e. $\mathcal{R}_\alpha \subseteq \mathcal{R}_{\alpha'}$).

### 4.6.4 Boyce-Codd Normal Form

A schema $\mathcal{R}$ is in **Boyce-Codd normal form (BCNF)**, if for every functional dependency $\alpha \to \beta$, at least one of the following two conditions holds:

- $\beta \subseteq \alpha$, i.e. the functional dependency is trivial.
- $\alpha$ is a superkey of $\mathcal{R}$.

The following **decomposition algorithm** takes a schema $\mathcal{R}$ as input and computes a lossless decomposition $\{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$, such that every $\mathcal{R}_i$ is in BCNF:

1. Start with $Z = \{\mathcal{R}\}$.
2. While there is a schema $\mathcal{R}_i \in Z$, that is not in BCNF:
   (a) Find a non-trivial functional dependency $\alpha \to \beta$ for $\mathcal{R}_i$, such that $\alpha \cap \beta = \emptyset$ and $\alpha \not\to \mathcal{R}_i$.
   (b) Decompose $\mathcal{R}_i$ into $\mathcal{R}_{i_1} := \alpha \cup \beta$ and $\mathcal{R}_{i_2} := \mathcal{R}_i \setminus \beta$.
   (c) Update $Z := (Z \setminus \mathcal{R}_i) \cup \{\mathcal{R}_{i_1}, \mathcal{R}_{i_2}\}$.

**Note:** The decomposition computed by this algorithm is in general not dependency preserving.

**Note:** BCNF implies 3NF

### 4.6.5 Fourth Normal Form

A schema $\mathcal{R}$ with the set of dependencies $D$ is in **fourth normal form (4NF)**, if and only if for all $\alpha \twoheadrightarrow \beta \in D^+$ at least one of the following conditions holds:

- $\alpha \twoheadrightarrow \beta$ is trivial, i.e $\beta \subseteq \alpha$ or $\beta = \mathcal{R} \setminus \alpha$.
- $\alpha$ is a superkey of $\mathcal{R}$.

**Note:** 4NF implies BCNF.

The decomposition algorithm for 4NF is analogous to the one for BCNF described in section **??**.

# 5 Database Internals

## 5.1 Database Optimizations

Many DBMS tasks, like load control and buffer management, are also carried out by the operating system. However the database has intimate knowledge of the workload an can predict the pattern of a query. In contrast, the operating system does generic optimizations. Sometimes the operating system overrides DBMS optimizations.

## 5.2 Storage Hierarchy

The storage of a database is organized in a hierarchy; different media ar combined to mimic one ideal storage. Higher layers of the hierarchy *buffer* data of lower layers by exploiting spatial and temporal locality of applications.

Furthermore the storage system can be distributed.

## 5.3 Buffer Management

A **buffer** is used to minimize disk accesses; pages are kept in main memory as long as possible. The buffer **replacement policy** defines which frame is chosen for replacement. Depending on the **access pattern** (sequential, hierarchical, random, cyclic) the policy can have a big impact on the number of I/O operations.

A page that is requested many times can be pinned by increasing the **pin count**. A page is a candidate for replacement if it is unpinned, i.e. the pin count is zero.

### 5.3.1 Least Recently Used

For each page in the puffer pool, the time of the last unpinning is kept track of. The **least recently used (LRU)** policy replaces the page with the oldest time.

**Note:** However, for sequential flooding $(1, 2, \ldots, n, 1, 2, \ldots)$ the most recently used policy works better.

### 5.3.2 Clock Replacement

The **clock replacement** policy is an approximation of LRU. It arrenges the frames into a cycle and stores a **reference bit** per page (One can think of this as a second chance bit). When the pin count of a page is reduced to zero, the reference bit is turned on.

When a replacement is necessary, the policy loops through the pages in the cycle and

- if the pin count of the page is zero and the reference bit is on, the reference bit is turned off and
- if the pincount is zero and the reference bit is off, the page is chosen for replacement.

## 5.4 Data Manager

TODO

## 5.5 Query Processing

There are three approaches to process a query:

- Compile a query into machine code. This approach achieves high performance.
- Compile a query into relational algebra and interpret the resulting expression. This is easier to debung and has better portability.
- Use a hybrid solution.

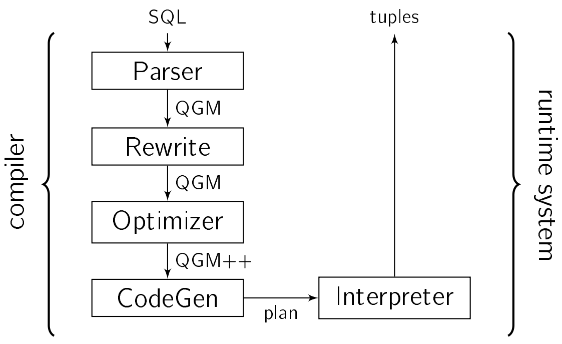

**Figure 9:** Query processor.

### 5.5.1 Parser

The **parser** generates a relational algebra tree for each subquery. Then it constructs a **query graph model (QGM)**, a graph of trees, where the nodes are subqueries and the edges represent the relationships between the subqueries.

### 5.5.2 Query Rewrite

### 5.5.3 Query Optimization

A query optimizer has two tasks: To determine the order of the operators and to determine a suitable algorithm for each operator. A cost model is applied to all alternative planse; the plan with lowest expected cost is selected.

]

**Example:** Consider the Cartesian product $A \times B \times C$, where $|A| = 1000$ and $|B| = |C| = 1$.

- The cost of computing $(A \times B) \times C$ is 2000,
- whereas the cost of $A \times (B \times C)$ is only 1001.

### 5.5.4 Iterator Model

The following methods are implemented by each **iterator**:

- **open()** Initialize the internal state; e.g. allocate buffer.
- **next()** produces the next tuple of the result.
- **close()** releases the buffer etc.

This way, a generic interface for operators is defined. Each operator can be implemented independently.

**Note:** Modern DBMS use a vector model, i.e next() returns a set of tuples.

## 5.6 Algorithms for Relational Algebra

There are many algorithms for relational algebra (scan, sort, join, group by, etc.). However, two specific algorithms are described in the following sections.

### 5.6.1 Two-Phase External Sort

If the data (input and result) does not fit into memory, **two-phase external sort** is needed to sort the tuples of a relation.

1. Load allocated buffer space with tuples and sort all tuples in the buffer pool. Write the sorted tuples (run) back to disk. Repeat this step, until all tuples are processed.
2. Use a heap to merge the tuples from all runs.

Let $n$ be the size of the input in pages and $m$ the size of the buffer in pages. If $m \geq n$, no merge is needed and if $m < \sqrt{n}$ multiple merge phases are necessary.

### 5.6.2 Grace Hash Join

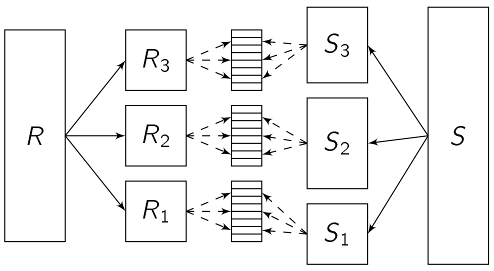

**Figure 10:** Grace hash join.

The **grace hash join** algorithms divides the relations $R$ and $S$ into partitions $R_1, \dots, R_n$ and $S_1, \dots, S_n$, using a hasfunction $h$. For all $1 \leq i \leq n$, every tuple $r \in R_i$ can only match tuples $s \in S_i$. Thus a hashtable $H_i$ (with another hash function $h'$) is used to join the tuples in $R_i$ and $S_i$.

```
GraceHashJoin($R$,$S$,$\alpha$)
  for all tuples $r \in R$ do
    add $r$ to $R_{h(r.\alpha)}$
  end for
  for all tuples $s \in S$ do
    add $s$ to $S_{h(s.\alpha)}$
  end for
  for all partitions $i \in \{1, \dots, n\}$ do
    build hastable $H_i$ for $R_i$
    for all tuples $s \in S_i$ do
      probe $H_i$ and output matching tuples
    end for
  end for
```

**Note:** The smaller relation $R$ is partitioned in such a way that every partition $R_i$ fits into main memory. This avoids rescanning the entire relation $S$.

### 5.6.3 Sorting vs Hashing

Both techniques, sorting and hashing, can be used for joins, grou-by etc. They have the same asymptotic complexity (in IO and CPU). However, sorting is more robust, but hashing has a better average case.

# 6 Transaction Processing

## 6.1 Transactions

A **transaction** $T_i$ consists of the following elementary operations:

- $b_i$ to indicate the beginning of a transaction (BOT),
- $r_i[x]$ to read from a data item $x$,
- $w_i[x]$ to write to a data item $x$,
- $a_i$ to perform an abort and
- $c_i$ to perform a commit.

Each transaction starts with a (often implicit) begin of transaction $b_i$ and ends with either an abort $a_i$ or commit $c_i$.

The order of the operations in $T_i$ defines a partial order $<_i$.

### 6.1.1 Properties of Transactions

The following four properties of a transaction are known as the **ACID principle**:

- **Atomicity:** A transaction is executed in its entirety or not at all.
- **Consistency:** A transaction executed in its entirety over a consistent database produces a consistent database.
- **Isolation:** A transaction executes as if it were alone in the system, regardless of other transactions.
- **Durability:** Commited changes of a transaction are never lost; i.e. they can be recovered after a system failure.

### 6.1.2 Savepoints and Backups

Defining a **savepoint** establishes a recoverable intermediate state. A **backup** transaction resets the state of the database to the most recent savepoint.

### 6.1.3 Conflicting Operations

Two operations are **conflicting** if uncontrolled cuncurrency causes potential inconcistency. Consider the possible operations of two transactions $T_i$ and $T_j$ that acces the same data item $A$:

- $r_i[x]$ and $r_j[x]$ are not conflicting, because both operations do not modify the database. Thus their order is irrelevant.
- $r_i[x]$ and $w_j[x]$ are conflicting, because, depending on their order $T_i$, is either reading the new or the old value.
- $w_i[x]$ and $r_j[x]$ is analogous to the previous case.
- $w_i[x]$ and $w_j[x]$ are conflicting, because the final state of the database depends on their order.

## 6.2 Cuncurrency Control Theory

### 6.2.1 Histories

A **history** $H$ for a set of transactions $\{T_1, \dots, T_n\}$ is a sequence of operations with a partial order $<_H$, such that:

- $H$ contains all operations of every transaction $T_i$, i.e. $H = \bigcup_{i=1}^{n} T_i$
- $<_H$ is compatible with all $<_i$, i.e $(\bigcup_{i=1}^{n} <_i) \subseteq <_H$.
- If $p, q \in H$ are conflicting operations, then either $p <_H q$ or $q <_H p$ holds.

$$\begin{tikzcd}
 & & r_2[x] \arrow[r] & w_2[y] \arrow[r] & w_2[z] \arrow[r] & c_2 \\
H = & r_3[y] \arrow[r] & w_3[x] \arrow[r] \arrow[u] & w_3[y] \arrow[r] \arrow[u] & w_3[z] \arrow[r] \arrow[u] & c_3 \\
 & r_1[x] \arrow[r] & w_1[x] \arrow[r] \arrow[u] & c_1 & &
\end{tikzcd}$$

**Figure 11:** A history for three transactions.

A history $H$ is **serial**, if for every two transactions $T_i$ and $T_j$ in $H$, either all operations from $T_i$ appear before all operations of $T_j$ or vice versa.

### 6.2.2 Equivalent Histories

Two histories $H_1$ and $H_2$ are equivalent, if and only if

- they are over the same transactions and contain the same operations and
- conflicting operations $p_i$ and $p_j$ of non aborted transactions are ordered in the same way in both histories, i.e $p_i <_{H_1} p_j$ holds if and only if $p_i <_{H_2} p_j$.

### 6.2.3 Serializable History

A history $H$ is **serializable** if and only if it is equivalent to a serial history $H_s$.

The **serializability graph** SG($H$) of a history $H$ over the transactions $T_1, \dots, T_n$ is a compact representation of the dependencies in $H$. For every pair of conflicting operations $p_i$ and $q_j$ in $H$ with $p_i <_H q_j$, the edge $(T_i, T_j)$ is added to the serializability graph SG($H$).

The **serializability theorem** states that a history is serializable if and only if its serializability graph SG($H$) is acyclic. A topological sort of SG($H$) is a serial history $H_s$ with $H \equiv H_s$.

**Example:** The serializability graph SG($H$) of the history $H$ depicted in figure **??** is:

$$\begin{tikzcd} & T_2 & \\ T_1 \arrow[ur] \arrow[rr] & & T_3 \arrow[ul] \end{tikzcd}$$

Thus the history $H_s = T_1 \mid T_3 \mid T_2$ is serial and equivalent to the history $H$.

## 6.3 Recovery Theory

### 6.3.1 Recoverable Histories

In a history $H$, a transaction $T_i$ is said to **reads** from another transaction $T_j$ if

- $T_i$ reads a data item $x$ that was written by $T_j$, i.e there are $r_i$ and $w_j$ with $w_j[x] <_H r_i[x]$,
- $T_j$ does not abort before the read, i.e. $a_j \not<_H r_i[x]$ and
- if there is a write $w_k[x]$ of another transaction with $w_j[x] <_H w_k[x] <_H r_i[x]$, then there is an $a_k <_H r_i[x]$.

A history $H$ is **recoverable** if for every transaction $T_i$ reads from another transaction $T_j$ the condition $c_j <_H c_i$ holds. An abort in a recoverable history does not affect already commited transactions.

### 6.3.2 Avoiding Cascading Aborts

A history $H$ **avoids cascading aborts (ACA)** if all transactions $T_i$ read only from committed transactions $T_j$, i.e $c_j <_H r_i[x]$.

### 6.3.3 Strict Histories

A history $H$ is **strict** if for all $w_j[x] <_H p_i[x]$, where $p_i$ is a read or write operation, either $a_j <_H p_i[x]$ or $c_j <_H p_i[x]$ holds.

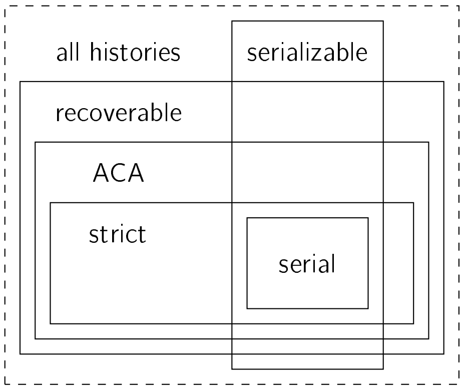

**Figure 12:** Classes of histories.

## 6.4 Database Scheduler

The task of a **database scheduler** is to order the operations of transactions $T_1, \ldots, T_n$ in such a way that the resulting history is reasonable. Serializability is often the minimal requirement. In general avoidance of cascading aborts is also required.

## 6.5 Lock Based Synchronization

Lock based synchronization is a widely used technique for schedulers. There are two sorts of locks:

- A **schared lock** $S$ is acquired if a transaction wants to read from an object.
- An **exclusive lock** $X$ is acquired if a transaction wants to write to an object.

### 6.5.1 Two-Phase Locking Protocol

The scheduler realizes serializability by following the **two-phase locking (2PL)** protocol. The following conditions must hold:

1. Every transaction must acquire a lock before accessing an object.
2. A transaction acquires a lock only once. Lock upgrades are possible.
3. A transaction is blocked if the lock requested cannot be granted.
4. Every transaction goes through two phases
   (a) Growth: Acquire locks, but never release a lock.
   (b) Schrink: Release locks, but never acquire a lock.
5. At the end of a transaction (commit or abort) all locks must be released.

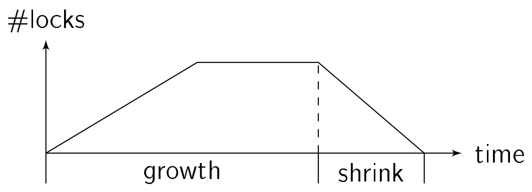

**Figure 13:** Visualization of the two-phase locking protocol.

### 6.5.2 Strict Two-Phase Locking Protocol

The **strict two-phase locking protocol** is like 2PL, except there is no shrink phase, i.e. all locks are kept until the end of the transaction.

**Note:** Strict 2PL avoids cascading aborts.

TODO(discussion of 2PL/ S2PL)

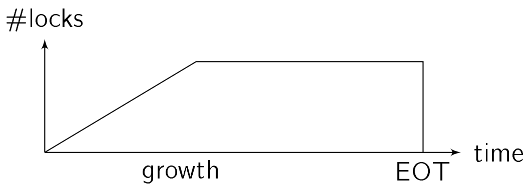

**Figure 14:** Strict two-phase locking protocol.

### 6.5.3 Deadlock Detection

A precise, but expensive, way to detect **deadlocks** is the **wait-for graph**: For every transaction $T_i$ that waits for another transaction $T_j$, the edge $(T_i, T_j)$ is added to the graph. There is a deadlock if and only if the wait-for graph has a cycle.

**Example:** A wait for-graph is depicted in figure **??**. Aborting transaction $T_2$ or $T_3$ resolves both cycles.

$$\begin{tikzcd}
T_1 \arrow[r] & T_2 \arrow[d] \arrow[dr] & \\
T_4 \arrow[u] & T_3 \arrow[l] & T_5 \arrow[l]
\end{tikzcd}$$

**Figure 15:** Wait-for graph with two cycles.

Another way to detect deadlocks is to set a **timeout** for every transaction's progress. If a timeout occurs, the system assumes a deadlock and resets the affected transaction.

A too short timeout leads to transactions being unnecessarily resetted, whereas a long timeout leads to prolonged deadlock situations.

## 6.6 Snapshot Isolation

In transaction processing, **snapshot isolation (SI)** is a guarantee that all reads made in a transaction will see a consistent snapshot of the database:

1. When a transaction $T_i$ starts, it receives a timestamp $\tau_i$.
2. All reads are carried out as of the database at time $\tau_i$. Historic versions of all objects are needed.
3. All writes are carried out in a separate buffer and become visible after a commit.
4. When a transaction $T_i$ with timestamp $\tau_i$ commits, the DBMS checks for conflicts and aborts $T_i$ if there is another transaction $T_j$ with timestamp $\tau_j$ that updated the same object and $\tau_i < c_j < c_i$.

### 6.6.1 Lost Updates

Figure **??** shows two histories that are accepted by snapshot isolation, but the update of transaction $T_2$ is lost.

| | $T_1$ | $T_2$ |
|---|---|---|
| 1 | | $b_2$ |
| 2 | | $r_2[x]$ |
| 3 | $b_1$ | |
| 4 | $r_1[x]$ | |
| 5 | | $w_2[x]$ |
| 6 | | $c_2$ |
| 7 | $w_1[x]$ | |
| 8 | $c_1$ | |

| | $T_1$ | $T_2$ |
|---|---|---|
| 1 | $b_1$ | |
| 2 | $r_1[x]$ | |
| 3 | | $b_2$ |
| 4 | | $r_2[x]$ |
| 5 | | $w_2[x]$ |
| 6 | | $c_2$ |
| 7 | $w_1[x]$ | |
| 8 | $c_1$ | |

**Figure 16:** Snapshot isolation and lost update.

### 6.6.2 Write Skew Anomaly

In a **write skew** anomaly, two transactions $T_1$ and $T_2$ concurrently read an overlapping data set (e.g. $x$ and $y$), concurrently make disjoint updates (e.g. $T_1$ updates $x$ and $T_2$ updates $y$, and finally commit, neither having seen the update performed by the other. In a serializable system, such an anomaly would be impossible.

### 6.6.3 Comparison with Two-Phase Locking

The history shown in figure **??** is accepted by both 2PL and SI. 2PL supports the serialization $T_2 \rightarrow T_3 \rightarrow T_1$ whereas SI enforces the serialization $T_1 \rightarrow T_2 \rightarrow T_3$, which may lead to inconsistency.

| | $T_1$ | $T_2$ | $T_3$ |
|---|---|---|---|
| 1 | $b_1$ | | |
| 2 | | $b_2$ | |
| 3 | | $w_2[x]$ | |
| 4 | | $w_2[y]$ | |
| 5 | | $c_2$ | |
| 6 | $r_1[x]$ | | |
| 7 | | | $b_3$ |
| 8 | | | $r_3[z]$ |
| 9 | | | $r_3[y]$ |
| 10 | | | $c_3$ |
| 11 | $w_1[z]$ | | |
| 12 | $c_1$ | | |

**Figure 17:** Interesting history.

- **Two-phase locking:** A lock for every read or write is needed. There is a total order of conflicting operations. Operations are not re-ordered and only serializable histories are allowed.
- **Snapshot isolation:** No read or write of a transaction is ever blocked; blocking only happens when a transaction commits. Aborts can be implemented very efficiently. There are no deadlocks, but unnecessary rollbacks. Non-serializable histories are allowed. Read-write conflicting operations are re-ordered and an abort is needed to deal with write-write conflicts.

## 6.7 Distributed Transaction Processing

### 6.7.1 Atomic Commit

An **atomic commit** enforces the following properties:

1. All processes that reach a decision reach the same one.
2. A process cannot reverse its decision.
3. A commit can only be decided if all processes vote yes.
4. If there are no failures and all processes voted yes, the decision will be to commit.
5. Consider an execution with normal failures. If all failures are repaired and no more failures occur for a sufficiently long time, then all processors will eventually reach a decision.

### 6.7.2 Two-Phase Commit Protocol

The **two-phase commit protocol (2PC)** is a distributed algorithm that coordinates the processes that participate in a distributed atomic transaction on whether to commit or abort the transaction.

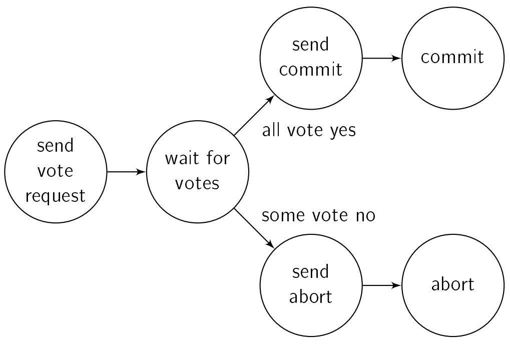

**Figure 18:** Two-phase commit coordinator.

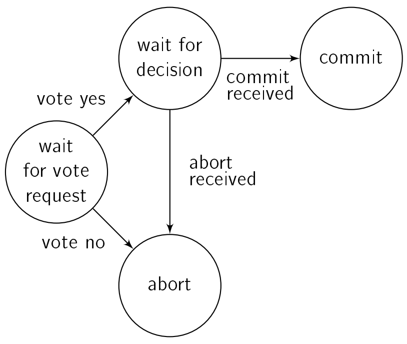

**Figure 19:** Two-phase commit participant.

1. The coordinator sends a *vote request* to all participants.
2. Upon receiving a *vote request*, a participant sends a message with yes or no. If the vote is no, the participant aborts the transaction and stops.
3. If the coordinator collects all votes and
   (a) sends *commit* if all participants voted yes or
   (b) sends *abort* if there is a participant that voted no.
4. A participant that receives a *commit* or *receive* from the coordinator decides accordingly and stops.7

If the coordinator times-out waiting fo votes, it can decide to abort. If a participant times-out waiting for a *vote request*, it can decide to abort. But if a participant times-out waiting for a decision it cannot decide anything; this state is called uncertainty period.

When in doubt, ask if any process has decided yet. If the coordinator fails after receiving all votes but before sending any *commit* message, all participants are uncertain and will not be able to decide anything until the coordinator recovers; they are blocked.

**Note:** 2PC meets the five atomic commit conditions.

### 6.7.3 Linear Two-Phase Commit Protocol

The **linear two-phase commit protocol** exploits a linear network topology to minimize the number of messages. The process $P_1$ starts the voting phase. If a process $P_i$ receives a *yes* and wants to commit, it sends a *yes* to process $P_{i+1}$. If the process $P_n$ receives a *yes* and wants to commit, a *commit* is sent back along all processes. If any process $P_i$ receives a *yes* and wants to abort, it sends an *abort* to $P_{i-1}$ and $P_{i+1}$. If a process receives an abort from one side, the abort is forwarded to the other side.

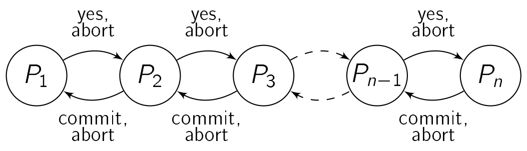

**Figure 20:** Message flow of the linear two-phase commit protocol.

The total number of messages sent by linear 2PC is at most $2n$ instead of $3n$. But the number of rounds is $2n$ instead of only 3.

### 6.7.4 Three-Way Commit Protocol

In the two-phase commit protocol there is the blocking situation in which all processes have voted yes but the coordinator fails. To avoid this situation the **three-phase commit protocol (3PC)** enforces the non-blocking rule: *No process can decide to commit if there are processes that are uncertain.*

TODO(WTF is this shit?)

[1]This state diagram was copied from the lecture slides; but I cannot explain the transition from *send ACK* to *abort*.

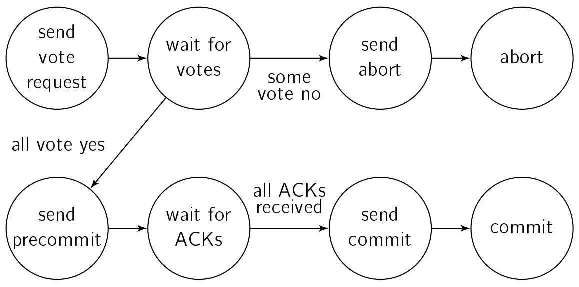

**Figure 21:** Three-way commit coordinator.

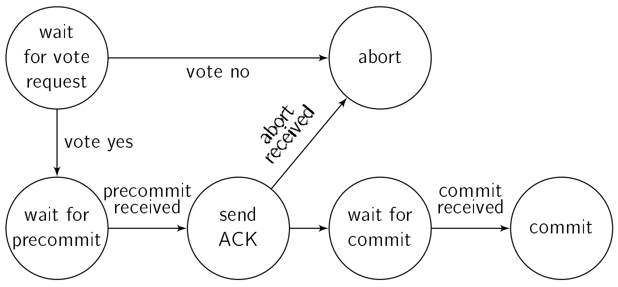

**Figure 22:** Three-way commit participant.[1]

# 7 Replication

Some reasons why **replication** is used are:

- **Performance:** Location transparency is difficult to achieve in a distributed environment. If everything is local, all accesses should be fast.
- **Fault tolerance:** If a site fails, the data it contains becomes unavailable. By keeping several copies, single site failures should not affect the overall availability.
- **Application type:** To avoid interference, databases have always tried to seperate queries from updates.

## 7.1 Replication Strategies

There are two basic parameters to select when designing a replication strategy.

- When to propagate the updates: synchronous (eager) or asynchronous (lazy).
- Where the updates can take place: primary copy (master) or update everywhere (group).

### 7.1.1 Synchronous Replication

**Synchronous replication** propagates any changes to the data immediately to all existing copies. Moreover, the changes are propagated within the scope of the transaction making the changes. The ACID properties apply to all copy updates.

- **Advantages:** There are no inconsistencies, i.e. all copies are identical. Thus reading the local copy yields the most up to date value. All changes are atomic.
- **Disadvantages:** The execution time and response time are longer, because a transaction has to update all sites.

### 7.1.2 Asynchronous Replication

**Asynchronous replication** first executes the updating transaction on the local copy. Then the changes are propagated to all other copies. While the propagation takes place, the copies are inconsistent.

- **Advantages:** A transaction is always local, i.e. the response time is short.
- **Disadvantages:** There are data inconsistencies. A local read does not always return te most up to date value. Changes to all copies are not guaranteed and the replication is not transparent.

### 7.1.3 Update Everywhere

With an **update everywhere** approach, changes can be initiated at any of the copies. That is, any of the sites which owns a copy can update the value of a data item.

- **Advantages:** Any site can run a transaction and the load is evenly distributed.
- **Disadvantages:** All copies need to be synchronized.

### 7.1.4 Primary Copy

With a **primary copy** approach, there is only one copy which can be updated (the master), all others (secondary copies) are updated reflecting the changes to the master.

- **Advantages:** No inter-site synchronization is necessary; it takes place at the primary copy. There is always one site that has all the updates.
- **Disadvantages:** The load at the primary copy can be quite large. Reading the local copy may not yield the most up to date value.

## 7.2 Replication Protocols

### 7.2.1 Quorum Protocols

**Quorums**

### 7.2.2 Deadlock Problem

### 7.2.3 Reconciliation

TODO

# 8 Database Security

## 8.1 Security Tasks

There are three kinds of security mechanisms in a DBMS:

- **Authentication:** Verifying the identification of a user.
- **Authorization:** Checking the access privileges.
- **Auditing:** Looking for violations (in the past).

Data security consists of confidentality and integrity.

## 8.2 Discretionary Access Controls

Access rules of **discretionary acces controls (DAC)** assign access rights $t$ for an object $o$ to a subject $s$. Formally, access rules are quintuples $(o, s, t, p, f)$, where

- $o \in O$ and $O$ is the set of **objects** (e.g. tables, tuples, attributes),
- $s \in S$ and $S$ is the set of **subjects** (e.g. users, processes, applications),
- $t \in T$ and $T$ is the set of **acces rights** (e.g. read, write, delete),
- $p$ is a predicate (e.g. level = 'c4') and
- $f$ is a boolean value specifying, whether $s$ may grant the privilege $(o, t, p)$ to another subject $s' \in S$.

A simple way to store the access rules is a so called acces matrix. Other ways to control accesses are views or query rewriting.

### 8.2.1 Access Control in SQL

```
grant update (id, name, semester)
   on student
   to peter;

create view first_semester as
   select *
   from student
   where semester = 1;
grant select
   on first_semester
   to tutor;
```

Personal records can be protected by aggregation.

# A SQL

The language **SQL (Structured Query Language)** is designed for managing data in relational DMBS. It is based upon relational algebra and tuple relational calculus. SQL is a family of standards:

- Data definition language (DDL)
- Data mnipulation language (DML)
- Query Language

**Note:** In SQL relations are referred to as tables and may contain duplicates.

## A.1 Data Types

Some of SQL's data types are:

- **char(*n*):** strings of length *n*, padded with spaces.
- **varchar(*n*):** strings of variable length with a maximum size of *n* characters.
- **integer:** for interger values.
- **numeric(*p*,*s*):** numbers with $p-s$ digits before the decimal and *s* digits after the decimal.
- **blob, raw**: large binaries.
- **clob**: large string values.
- **date**: dates.

## A.2 Data Definition Language

A **data definition language (DDL)** is a syntax for defining or modifying data structures, especially database schemas. The following SQL statement illustrates, how a new table is created:

```
create table student
  (id   integer      primary key,
   name varchar(20)  not null),
   age  integer;
```

Columns of an existing table can be added and altered:

```
alter table student
   add column semester integer;
alter table student
   alter column name varchar(30);
```

Similarly, a column is removed with **drop column**. Indexes is created and deleted like this:

```
create index my_index on
   student(name, age);
drop index my_index;
```

## A.3 Data Manipulation Language

A **data manipulation language (DML)** is used to insert, update and delete tuples:

```
insert into student (id, name)
   values (123, 'L.␣Skywalker');
update student
   set semester = semester+1;
delete student
   where age < 13;
```

## A.4 Queries

```
select [distinct] columns
   from tablename [as alias]
   [where condition]
   [group by (attribute)+]
   [having condition]
   [order by (attribute [asc|desc])+]
```

- The **distinct** keywords eliminates all duplicate tuples.
- Results are sorted using the **order by** key words.

| *v* | **not** *v* |
|---|---|
| true | false |
| unknown | unknown |
| false | true |

| **and** | true | unknown | false |
|---|---|---|---|
| true | true | unknown | false |
| unknown | unknown | unknown | false |
| false | false | false | false |

| **or** | true | unknown | false |
|---|---|---|---|
| true | true | true | true |
| unknown | true | unknown | unknown |
| false | true | unknown | false |

## A.5 Isolation in SQL92

Isolation in a database determines how and when changes of a transaction becom visible for other transactions.

```
set transaction
   [read only, | read write,]
   [isolation level
      read uncommited,   |
      read commited,     |
      repeatable read,   |
      serializable ,]
  [diagnostic size  ... ,]
```

- **Read uncommited:** This is the lowest level of isolation and is only allowed for read only transactions. Reads do not need a shared lock and are not hindered by exclusive locks.
- **Read commited:** Transactions only read commited versions of objects. However, one transaction may read different versions of an object.
- **Repeatable read** Reading different versions of the same object is prevented, but phantoms can still happen.
- **Serializable:** Full isolation is guaranteed.