CS348: Introduction to Database Management

Helena S. Ven

19 June 2019

Instructor: David Toman (david@uwaterloo.ca)

Office:

Text: *Database System Concepts (7th Edition)*

Topics:

There are 4 assignments throughout the term.

Final: NoSQL Spark, Hadoop are not covered on final exams.

Coverage upto section 11 (11/12–14)

Index

Caput 1

Introduction

The course is designed to meet the needs of students who are interested in using database technology in system development.

We attempt to answer the questions:

1. Why do we use databases? What are the functionalities provided by a database management system.
2. How do we use a Database Management System?
3. How do we design a database?

1.1 Data

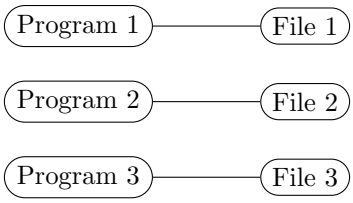
Definition

The ANSI definition of **data** is:

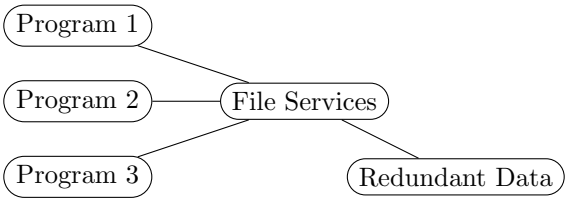
1. A representation of *facts*,*concepts*, or *instructions* in a formalised manner suitable for communication, interpretation, or processing by humans or by automatic means.
2. Any representation such as characters or analog quantities to which *meaning is or might be assigned*. Generally, we perform operations on data or data items to supply some information about an entity.

A data is **persistent** if it is preserved after the storage system shuts down.

In early stage of data management, data is stored per program, which leads to inefficiencies:



Using a database approach, multiple programs can access the same piece of data, and it frees the program developers of the hassle of managing data encodings in the hard drive.



Definition

A **database** is a large and persistent collection of factual data and metadata organised in a way that facilitates **efficient retrieval** and **revision**.

Example of data: John's age is 42.

Example of metadata: There is a concept of an employee that has a name and an age. Metadata is data about data.

Most programming languages allow for defining a structured type, e.g.

```
struct Instructor
{
  char id[5];
  char name[2];
  char dept_name[20];
  int salary;
};
```

This defines a new *record type* called `Instructor` with four fields, each has a associated name and type. A university organisation may have several record types.

1.1.1 Database Management

Definition

A **data model** determines the nature of the metadata and how retrieval and revision is expressed.

A **database management system** (**DBMS**) is a program or set of programs that implements a data model. A DBMS has

1. Management an underlying data model.
2. Access control
3. Concurrency control: Multiple concurrent users can access data.
4. Database recovery: Reliability
5. Database maintenance: Revising metadata.

Example of databases: A file cabinet, A library system, An inventory control system.

From textbook: A **database management system** (DBMS) is a collection of interrelated data and set of programs to access the data. The set of data is usually referred to as **database**.

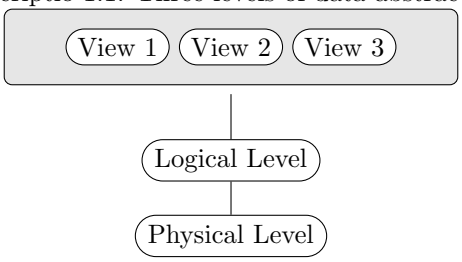
Databases are widely used in:

- Enterprise information
- Banking and finance
- Universities, Airlines, Telecommunication, etc.

Earlier methods to compuerise data is to store it in files. The *file processing system* used to manage these files is supported by a conventional operating system. This system has some disadvantages:

- **Data redundancy** and **inconsistency**: When time goes by the various files in the database may have different formats. Information maybe duplicated among several files. This sometimes leads to the various copies of data being different.
- **Difficulty of Access**: If the designer of the file processing system did not anticipate customised query requests, the query may have to be done manually or the management program needs to be changed.

Descriptio 1.1: Three levels of data abstraction



- **Isolation**: Data are scattered in various files and maybe in different formats, so writing new application programs to query the appropriate data is difficult.
- **Integrity probelms**: The data values stored in the database must satisfy some *consistency constraints*. (e.g. for accounting purposes) It is difficult to change the programs to enforce them.
- **Atomicity problem**: If the management program fails, the data must be restored to a consistent state that existed prior to the failure. e.g. A instruction that transfers funds from one account to another must not change the total amount of money in the database.
- **Concurrent-access anomalies**: Multiple concurrent updates may result in inconsistent data.
- **Security problems**: Not every user of the database system should be able to access all of the data.

Since the database system users are not computer trained, developers hide the complexity through several levels of abstraction:

- **Physical**: The lowest level describes how the data are actually stored. The physical level describes complex low-level data structures in detail.

  In the compiled program (physical level), the `Instructor` record can be described as a block of consecutive storage locations but this fact is hidden from the programmer.
- **Logical**: This level describes *what* data are stored in the database and what relationships exist among the data. The logical level describes the entire database in terms of a smaller number of relatively simple structures. The users of the logical level does not need to be aware of this complexity and this is referred to as *physical data independence*.

  At the logical level, `Instructor` is defined by a type definition as in the previous code segment.
- **View**: The highest level of abstraction describes only part of the database. Even though the logical level uses simpler structure, complexity remians because of the variety of information stored in a large database.

  Many users of the database system does not need all of the information but they need to access only a part of the database. The view provides the mechanism for user privilege.

1.2 Instances and Schemas

Databases change overtimea as information is inserted and deleted.

- The collection of information stored in the database at a particular moment is the **instance**.
- The overall design of the database is the **schema**, which is changed infrequently if at all.

Definition

A database **schema** is a collection of metadata conforming to an underlying data model. (Signatures and Integrity Constraints)

A database schema corresponds to the variable declarations in the program. Each variable has a particular value at a instance and the values of the variable sin a program at a point correspond to an **instance** of a database schema.

Database systems have several schemas, *physical schema*, *logical schema*, and sometimes some schemas, *subschemas* at the view level. Application programs exhibit **physical data independence** if they do not depend on the physical schema.

1. A **external schema** (view) is what the application programs and users see. It may differ for different users of the same database.
2. A **conceptual schema** is a description of the logical structures of all data in the database.
3. A **physical schema** is a description of physical aspects.

A **data model** is a collection of conceptual tools describing data, relationships, semantics, and consistency. There are four categories of data models:

- **Relational**: Use a collection of tables to represent both data and relationships among those data. The tables are **relations**. This is a *record-based* model since the database is structured in fixed-format records of several types.
- **Entity-Relationship Model**: Uses a collection of basic objects, *entities*, and *relationships* among thees objects.
- **Object-Based Data Model**: Exenting the E-R model with encapsulation, methods, and object identity. The object-relational data model combines features of the object-oriented data model and relational data model.
- **Semistructured Data Model**: Permit the specification of data where individual data items of the same type may have different sets of attributes.

  The **Extensible Markup Language** (XML) is widely used to represent semistructured data.

Historically, the **network data model** and **hierarchical data model** preceded the relational data model. These models were tied closely to the underlying implementation. They are mostly obsolete.

A database system provides

- A **Data Definition Language** (DDL) to specify the schema.

  A special type of DDL, **data storage and definition** language, specifies the storage structure and access methods. These implementation details are usually hidden from the users.
- A **Data Manipulation language** (DML) to express database queries and updates. It enables users to access or manipulate data as organised by the appropriate data model. The types of access are:

  1. Retrieval of information
  2. Insertion of new information,
  3. Deletion of information
  4. Modification of information

  There are two types:

  – Procedural: Require a user to specify what data are needed and how to obtain the data.

  – Declarative: Require a user to specify what data are needed without specifying how to obtain. In this case the database needs to decide how to obtain the data.

In practice, they are not separate language and instead form parts of a single database language, such as SQL .

A **queury** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is the **query language**. Although technically incorrect, it is common practice to use *query language* and *data-manipulation language* synonymously.

1.2.1 Consistency constraints

The data values stored in the database must satisfy certain *consistency constraints*. For example, suppose the unviersity require that the account belance never become negative. DDL provides a facility to specify such a constraint. The database checks these constraints every time the database is updated. In general a constraint can be an arbitrary predicate pertaining to the database but some may be costly to test.

- **Domain Constraint**: A domain of possible value must be associated with every attribute. This is the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered.
- **Referential Integrity**: We wish to ensure that a value that appears in one relation for a given set of attributes also appear in a certain set of attributes in another relation.

  For example, the department listed for each course must be one that actually exists.
- **Assertions**: An assertion is any condition that the database must always satisfy. e.g. Every department must have at least five courses offered every semester. The system tests for an assertion when it is created. Modifications to a instance is only allowed if it satisfies all assertions.
- **Authorisation**: We may want to differentiate among the users. The users may have several privileges: Read, Insert, Update, Delete.

The DDL gets as input some instructions and generates some output. The output of the DDL is place in the data dictionary, which contains metadata. (i.e. data about data). The data dictionary is a special type of table that can only be accessed and updated by the database system itself.

Caput 2

Relational Databases

A **relational database** is based on the relational model and uses a collection of tables to represent both data and the relationships among those data.

2.1 Relational Calculus

The **set-comprehension notation** in mathematics allows us to describe all elements of a set satisfying a condition:

$$\{(x_1, \ldots, x_n) : \text{condition}\}$$

Set comprehension does not specify how to find individual $(x_1, \ldots, x_n)$'s but only what $(x_1, \ldots, x_n)$'s are required. For example, the set below describes all natural numbers that add to 5.

$$\begin{aligned}&\{(x, y) : x + y = 5\}\\ &= \{(0, 5), (1, 4), (2, 3), (3, 2), (4, 1), (5, 0)\}\end{aligned}$$

The relation $x + y = 5$ can also be written as $+(x, y, 5)$, where the $+$ table is defined as

		+
0	0	0
0	1	1
1	1	2
	...	

The query below finds x, y with $x + y = x - y$.

$$\{(x, y) : \exists z. + (x, y, z) \wedge +(z, y, x)\}$$

The query below finds the additive identity:

$$\begin{aligned}&\{x : +(x, x, x)\}\\ &= \{0\}\end{aligned}$$

In relational databases, we organise all data in (a finite number of) relations. This allows for:

1. Simple and clean data model
2. Powerful and declarative query/update
3. Semantic integrity constraint
4. Data independence

A relational database has components:

1. Universe: A set of values $\mathbf{D}$ with equality $=$
2. Relation: Predicate name R and arity (number of columns) k. An relation is an instance $\mathbf{R} \subseteq \mathbf{D}^k$.
3. Database: A signature $\rho = (R_1, \ldots, R_n)$ of predicate names. An **instance** is $\mathbf{DB} = (\mathbf{D}, =, \mathbf{R}_1, \ldots, \mathbf{R}_n)$ where each $\mathbf{R}_i$ is a instance of R_i.

For example, if all the integers are stored in a database, we have $\rho = (P, T)$ and $\mathbf{DB} = (\mathbb{Z}, =, \mathbf{P}, \mathbf{T})$.

Definition

The **truth** of a formula φ is defined with respect to:

- A *database instance* $\mathbf{DB}$
- A *valuation* $\theta : \{x_1, \ldots\} \to \mathbf{DB}$

Using the rules

$$\begin{aligned}
&\mathbf{DB}, \theta \models R(x_1, \ldots, x_k) && \text{if } R \in \rho, (\theta(x_1), \ldots, \theta(x_k)) \in \mathsf{R} \\
&\mathbf{DB}, \theta \models x_i = x_j && \text{if } \theta(x_i) = \theta(x_j) \\
&\mathbf{DB}, \theta \models \varphi \wedge \psi && \text{if } \mathbf{DB}, \theta \models \varphi \text{ and } \mathbf{DB}, \theta \models \psi \\
&\mathbf{DB}, \theta \models \neg\varphi && \text{if not } \mathbf{DB}, \theta \models \varphi \\
&\mathbf{DB}, \theta \models \exists x_i.\varphi && \text{if } \mathbf{DB}, \theta[x_i \mapsto v] \models \varphi \text{ for some } v \in \mathbf{D}
\end{aligned}$$

2.1.1 Queries

Definition

A **query** in the relational calculus is a set comprehension of the form

$$\{(x_1, \ldots, x_k) : \varphi\}$$

An **answer** to a query $\{(x_1, \ldots, x_k) | \varphi\}$ on $\mathbf{DB}$ is the relation

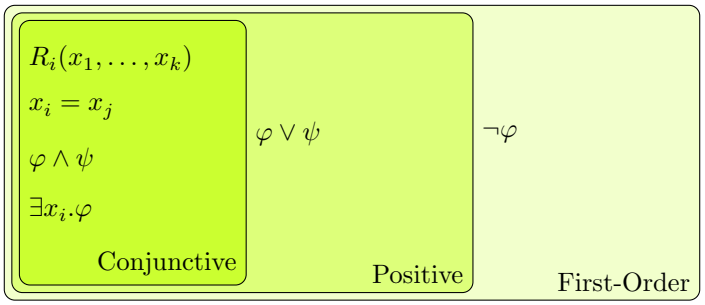
$$\{(\theta(x_1), \ldots, \theta(x_k)) : \mathbf{DB}, \theta \models \varphi\}$$

where $\{x_1, \ldots, x_k\} = \mathrm{FV}(\varphi)$ is the set of **free variables** of φ.

A formula that has no free variables is a **sentence**.

When we query a condition $C(x, y)$, $C(x, y)$ will be true for any *valuation* $\{x \mapsto a, y \mapsto b, \ldots\}$ exactly when $(a, b) \in \mathbf{C}$.

From simple queries we can build more complex ones using logical connectives i.e. Conjunction, Disjunction, Negation. We can also use quantifiers such as $\exists$.



Example:

The following query represents publications with at least one author.

$$\{x : (\exists y.\mathsf{PUBLICATION}(x, y)) \wedge (\exists z, w.\mathsf{WROTE}(z, x) \wedge \mathsf{AUTHOR}(z, w))\}$$

This query can be slightly optimised:

$$\{x : (\exists y.\mathsf{PUBLICATION}(x, y)) \wedge (\exists z, w.\mathsf{WROTE}(z, x))\}$$

Modern database query languages can often do this optimisation implicitly.

The following query generates all possible pairs of distinct authors.

$$\{(x, y) : (\exists w, z.\mathsf{AUTHOR}(w, x) \wedge \mathsf{AUTHOR}(z, y)) \wedge \neg(w = z)\}$$

We can add a clause to find pairs of distinct authors which wrote the same title:

$$\{(x, y) : (\exists w, z.\mathsf{AUTHOR}(w, x) \wedge \mathsf{AUTHOR}(z, y)) \wedge (\exists v.\mathsf{WROTE}(w, v) \wedge \mathsf{WROTE}(z, v))\}$$

If we want to insist that the author is unique:

$$\left\{x : (\exists y.\mathsf{PUBLICATION}(x, y)) \wedge (\exists z, w.\mathsf{WROTE}(z, x) \wedge \forall z'.\mathsf{WROTE}(z', y) \rightarrow z = z')\right\}$$

We can convert this to an expression using standard logic symbols with

$$\wedge\forall z'.\mathsf{WROTE}(z', y) \rightarrow z = z' \equiv \neg\exists z'.\mathsf{WROTE}(z', y) \wedge z \neq z'$$

A **query reduction** reduces one query to a simpler one. For example, the definition of additive identity in PLUS is

$$\{x : \forall y.\mathsf{PLUS}(x, y, y) \wedge \mathsf{PLUS}(y, x, y)\}$$

But we know a simpler query resulting in the same output:

$$\{x : \mathsf{PLUS}(x, x, x)\}$$

To reduce the first expression to the second, we need additional knowledge of the natural numbers and PLUS.

1. By commutativity $\forall x, y, z\, \mathsf{PLUS}(x, y, z) \rightarrow \mathsf{PLUS}(y, x, z)$, so
   $\mathsf{PLUS}(x, y, y) \equiv \mathsf{PLUS}(y, x, y)$
2. Addition is a binary function, so $\forall x, y, z_1, z_2\, \mathsf{PLUS}(x, y, z_1) \wedge \mathsf{PLUS}(x, y, z_2) \rightarrow z_1 = z_2$
3. Addition is a total function: $\forall x, y.\exists z\, \mathsf{PLUS}(x, y, z)$
4. By monotonicity, $\mathsf{PLUS}(x, y, y) \equiv \mathsf{PLUS}(x, x, x)$

Queries are set comprehensions with formulae in First-Order logic.

2.1.2 Integrity Constraints and Domain Independence

A relational *signature* captures only the structure of relations. For example, the publication database does not enforce that each book must have at least 1 author. Valid database instances satisfy additional integrity constraints.

Examples:

- Values of a particular attribute belong to a data type
- Values of attributes are unique among tuples in a relation (i.e. keys)
- Values appearing in one relation must also appear in another (referential integrity)

- Values cannot appear simultaneously in certain relations (disjointness)
- Values in certain relation must appear in at least one of another set of relations (coverage)

Definition

A **integrity constraint** is a closed formula in First-Order logic.

Note. A database is *finite*. This fact cannot be expressed in first-order logic. Therefore we can have *unsafe* (i.e. unbounded) queries

- $\{(y) : \neg\exists x.\mathsf{AUTHOR}(x, y)\}$: This asks for the set all strings which are not book titles.
- $\{(x, y, z) : \mathsf{books}(x, y, z) \vee \mathsf{proceedings}(x, y)\}$: The landmine in this formula is the disjunction $\vee$. An infinite number of z's can satisfy this formula.
- $\{(x, y) : x = y\}$: The "table" of $=$ is infinite.

Definition

A query $\{\boldsymbol{x} : \varphi\}$ is **domain-independent** if

$$\mathsf{DB}_1, \theta \models \varphi \iff \mathsf{DB}_2, \theta \models \varphi$$

For any pair of database $\mathsf{DB}_i \coloneqq (\mathbf{D}_i, =, \mathbf{R}_1, \ldots, \mathbf{R}_k)$ and valuation θ.

Domain independent and finite database gives safe queries.

2.1.1 Theorem.
Answers to domain-independent queries contain only values that exist in $\mathbf{R}_1, \ldots, \mathbf{R}_k$ *(the active domain).*

Proof. Suppose $Q \coloneqq \{\boldsymbol{x} : \varphi\}$ is domain-independent. Then if $y \in Q \setminus (\mathbf{R}_1 \cup \cdots \cup \mathbf{R}_k)$, we can redefine the domain to be $\mathbb{D}' \coloneqq \mathbb{D} \setminus \{y\}$ and under this domain $y \notin Q$, so this is a contradiction and y cannot exist. □

2.1.2 Theorem.
Satisfiability of first-order formulae is undecidable.

Proof. Reduction from PCP. □

2.1.3 Theorem.
Domain-independence of first-order queries is undecidable.

Proof. Consider queries of the form $Q \coloneqq \{x : (x = x) \wedge \varphi\}$ where x does not appear in φ.
If φ is unsatisfiable, then no valuation θ exists such that

$$\mathbf{DB}, \theta \models \varphi$$

and thus $Q = \varnothing$ is domain-independent.

If φ is satisfiable, there exists a valuation θ such that the free variables in φ are assigned so that φ is true. For this θ, Q will contain all of the elements of domain $\mathbf{D}$ and therefore Q is domain-dependent.

Since proving domain-dependence is equivalent to proving satisfiabilty, domain-dependency is undecidable. □

Definition

A formula φ is **range-restricted** if φ has the form (where φ_i are also range restricted):

$$
\begin{aligned}
&R(x_{i_1}, \ldots, x_{i_k}) && \\
&\varphi_1 \wedge \varphi_2 && \\
&\varphi_1 \wedge (x_i = x_j) && (\{x_i, x_j\} \cap \mathrm{FV}(\varphi_1) \neq \varnothing) \\
&\exists x_i . \varphi_1 && (x_i \in \mathrm{FV}(\varphi_1)) \\
&\varphi_1 \vee \varphi_2 && (\mathrm{FV}(\varphi_1) = \mathrm{FV}(\varphi_2)) \\
&\varphi_1 \wedge \neg \varphi_2 && (\mathrm{FV}(\varphi_2) \subseteq \mathrm{FV}(\varphi_1))
\end{aligned}
$$

There are some formulae which are not range restricted but still domain independent, such as

$$P(x, y) \wedge (Q_1(x) \vee Q_2(y))$$

but we can write this as

$$(P(x, y) \wedge Q_1(x)) \vee (P(x, y) \wedge Q_2(y))$$

2.1.4 Theorem.
Any range restricted formula is domain independent.

2.1.5 Theorem.
Any domain independent query can be written equivalently in range restricted form.

Proof. Let φ be a domain independent query. Then:

1. Restrict every variable in φ to the active domain.
2. Express the active domain using a unary query on the database.

□

2.2 Tables

Each **table** in a relational database has multiple columns and each column has a unique name.

For example, the following tables are defined as `instructor(id, name, deptname, salary)` and `department(deptname, building, funding)`

`instructor` Table

`id`	`name`	`deptname`	`salary`
1001	Marcoux, Laurent	Mathematics	50000
1242	Wu, Changbao	Statistics	40000
1243	Hofert, Marius	Statistics	40000
3245	Jayasundra, Rohan	Physics	84000
3291	Strickland, Donna	Physics	74000

`department` Table

`deptname`	`building`	`funding`
Statistics	MC	100
Mathematics	DC	80
Physics	PHY	3000

A row in the table represents a *relationship* of a set of values. Each table thus corresponds to a mathematical *relation*. A *tuple* is a list of values. **The orders in which the tuples appear in a relation is irrelevant**.

Each attribute in the relation has a **domain** which is a set of all permissible values for this attribute. e.g. The domain of `name` is the set of possible instructor names. A domain is **atomic** if elements of the domain are considered to be indivisible units. For example, if the `instructor` table has a `phone_numbers` attribute representing the set of phone numbers of the instructor, the domain of `phone_numbers` is not atomic. For now we shall assume that all attributes have atomic domains.

The **null** value is a special value that indicates a value is unknown or does not exist.

Definition

A **relational datbase schema** is a signature ρ and a (finite) set of integrity constraints Σ on ρ.
A instance **DB conforms** to a schema Σ, denoted

$$\mathbf{DB} \models \Sigma$$

if $\mathbf{DB}, \theta \models \varphi$ for any integrity constraint $\varphi \in \Sigma$ and any valuation θ.

2.3 Data Complexity

Evaluation of every query must terminate. (relational calculus is not Turing complete).

Definition

Data Complexity in the size of the database for a fixed query is in

- PTIME
- LOGSPACE
- AC_0: Constant time on polynomially many CPU's in parallel.

Combined Complexity is

- in PSPACE
- Can express NP-hard problems such as SAT.

Definition

Query Evaluation is the process of finding all answers to a query in a database.
Query Satisfiability is to determine if there is a finite database **DB** for which the answer of the query is non-empty.

Query satisfiability is much harder and can be solved for fragments of the query language.

Definition

A query $\{(x_1, \ldots, x_k) : \varphi\}$ **subsumes** a query $\{(x_1, \ldots, x_k) : \psi\}$ w.r.t. database schema Σ if

$$\{(\theta(x_1), \ldots, \theta(x_k)) : \mathbf{DB}, \theta \models \psi\} \subseteq \{(\theta(x_1), \ldots, \theta(x_k)) : \mathbf{DB}, \theta \models \varphi\}$$

for every datbase **DB** s.t. $\mathbf{DB} \models \Sigma$.

There are queries unexpressible in relational calculus because RC is not Turing complete. e.g.

- Ordering, arithmetic, string operations
- Counting, Aggregation
- Reachability, Connectivity (Paths in graph)

2.4 Keys

Definition

A **superkey** is a set of one or more attributes that, taken collectively, allows unique identification of tuples in a relation.

Let R denote the set of attributes in schema of relation r. If for any distinct tuples t_1, t_2, the subset K of R has $t_1[K] \neq t_2[K]$, then K is a *superkey*.

For example, the `id` attribute of the `instructor` relation is sufficient to distinguish one `instructor` tuple from another. The `name` attribute is not a superkey since two instructors may have the same name.

Note that superkeys may contain extraneous attributes. For example, if K is a superkey, then any $K' \supseteq K$ is a superkey. A minimal superkey (i.e. one with no extraneous attributes) is a **candidate key**.

It is possible that several distinct candidate keys exist. For example, if two instructors in the same department do not have the same name, the combination of `name` and `deptname` is sufficient to distinguish among members of `instructor`.

A **primary key**[1] is a candidate key that is chosen by the database designer as the principal means of identifying tuples in a relationship. This primary key must be chosen to eliminate all possibilities of collision. e.g. The name of a person is not a primary key.

2.4.1 Foreign Keys

A relation r_1 may include the primary key of another relation r_2. This attribute is a **foreign key** from r_1 referencing r_2. r_1 is the **referencing relation** and r_2 is the **referenced relation**.

Each `instructor` tuple has a associated `deptname` attribute. We demand that this `deptname` references a valid department in the `department` table. This is a **referential integrity constraint**.

Relationships between objects that are present in an instance are true. The relationships absent are false. For example, $(1001, \text{L. Marcoux}, \text{Mathematics}, 50000) \in$ `instructor`, so it is true that L. Marcoux is an instructor in Mathematics.

2.5 Database Design

Database systems are designed to manage large bodies of information. Designing a database mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of an enterprise requires attention to a broader set of issues.

The phases of database design:

1. Characterise the data needs of the prospective database user.
2. (Conceptual-design) Choose a data model and by applying concepts of the chosen data model, translates the requirements into a conceptual schema.

   Determine what attributes are captured in the database and how to group the attributes into various tables.
3. (Logical design phase) Designer maps high-level conceptual schema onto the implementation data model.
4. (Physical design phase) Physical features of the database are specified.

To illustrate the design process, consider a database for a university could be designed. The initial specification of user requirements may be based on interviews with the database users.

- The university is organised into departments, each identified by a unique name, located in a building, and has a funding.
- The department has a list of courses it offers. Each associated with a course id, title, department name, and credits, and may also have associated prerequisites.
- Instructors are identified by their unique id. Each instructor has a name, department, and salary.

[1] It is customary to list the primary key attributes before the other attributes

- Students are identified by their unique id. Each student has a name, an associated major department, and total credits.
- The university maintains a list of classrooms, specifying the building, room number, and capacity.
- The university maintains a list of all classes taught.
- The department has a list of teaching assignments specifying the section the instructor is teaching.
- The university has a list of all student course registrations.

The **Entity-Relationship Model** uses a collection of basic objects, *entities*, and *relationships* among these objects. Persons or bank accounts can be entities.

2.5.1 Schema Diagrams

A database along with primary key and foreign key dependencies, can be depicted using **schema diagrams**:

1. Each relation appears as a box.
2. The title of the box is the name of the relation.
3. The attributes are listed in a box.
4. Primary key attributes are underlined.
5. Foreign key dependencies appear as arrows from the referencing relation to the primary key of the referenced relation.

2.6 Relational Algebra

A **query language** is a language in which a user requests information about the database. A query language is either **procedural**, in which the user instructs the system to perform a sequence of operations on the database, or **non-procedural**, in which the user describes the target information without giving a specific procedure for obtaining the information.

All procedural relational query languages provide a set of operations that can be applied to either a single relation or a pair of relations, which always result in a single relation. This property allows one to combine several of these operations in a modular way. Including:

- **Selection**: Filter of specific tuples from a single relation that satisfies some particular *predicate*. The result is a new relation that is a subset of the original relation.
- **Join**: Combine two relations by merging pairs of tuples from relations into one single tuples. There are several ways to join relations.
- **Natural join**: In this scheme a tuple from the `instructor` relation matches a tuple in the `department` if the values of their primary key, `deptname`, matches:

`instructor` Table

id	name	deptname	salary	building	funding
1001	Marcoux, Laurent	Mathematics	50000	DC	80
1242	Wu, Changbao	Statistics	40000	MC	100
1243	Hofert, Marius	Statistics	40000	MC	100
3245	Jayasundra, Rohan	Physics	84000	PHY	3000
3291	Strickland, Donna	Physics	74000	PHY	3000

- **Cartesian Product**: Create a relation that consists of all possible tuple pairs from the two tuples. The resulting table size is the product of the individual table sizes.
- **Union**, **Intersection**, **Difference**: Set operations can be performed on two "similarly structured" tables.

Definition

Relational Algebra is a set of operations on relations.

1. σ (Selection): $\sigma_P(r)$
   Return tuples that satisfy the predicate P.
2. Π (Projection): $\Pi_{a_1,\ldots,a_n}(r)$
   Return specified columns from the relation.
3. $\bowtie$[a] (Natural join): $r \bowtie s$
   Return pairs of rows that have the same values on all attributes with the same name.
4. $\times$ (Cartesian product): $r \times s$
5. $\cup$ (Union):
   Output the union of tuples.
6. $\setminus$ (Difference)

[a] This symbol is a *bowtie*

Relational algebra defines a *set of operations* on the universe of finite relations.

Definition

The **projection operator** Π_V is defined as

$$\Pi_V(r) \coloneqq \{(x_{i_1}, \ldots, x_{i_k}) : (x_1, \ldots, x_n) \in r, i_j \in V\}$$

Definition

The **selection operator** σ_φ is define as

$$\sigma_\varphi(r) \coloneqq \{(x_1, \ldots, x_n) \in r : \varphi(x_1, \ldots, x_n)\}$$

Definition

The **Cartesian product** of two relations is

$$r \times s \coloneqq \{(x_1, \ldots, x_n, y_1, \ldots, y_m) : (x_1, \ldots, x_n) \in r, (y_1, \ldots, y_m) \in s\}$$

2.6.1 Theorem (Codd).
Every domain independent relational calculus query has an equivalent relational algebra expression.

$$\begin{aligned} R(x_1, \ldots, x_k) &\equiv r \\ Q \wedge x_i = x_j &\equiv \sigma_{\cdot_i = \cdot_j}(q) \\ \exists x_i . Q &\equiv \Pi_{\mathrm{FV}(Q) - \{\cdot_i\}}(q) \\ Q_1 \wedge Q_2 &\equiv q_1 \times q_2 \\ Q_1 \vee Q_2 &\equiv q_1 \cup q_2 \\ Q_1 \wedge \neg Q_2 &\equiv q_1 \setminus q_2 \end{aligned}$$

Queries in $\wedge$ must have disjoint sets of free variables.

2.7 Transactions

When multiple applications access the same data concurrently, racing condition problems can occur. If there is a global database lock to prevent this from happening, the database becomes unacceptably slow. Is there a way to improve upon this?

Each application may think it is the sole application accessing the database. The DBMS needs to ensure this.

The following properties are ensured by the DBMS:

1. **Atomicity**: A transaction occurs either entirely, or not at all.
2. **Consistency**: Each transaction preserves the consistency of the database.
3. **Isolated**: Concurrent transactions do not interfere.
4. **Durable**: Once completed, a transaction's changes are permanent.

Types of database users:

1. End user: Accesses database indirectly through forms.
2. Application Developer
3. Database Administrator

Caput 3

SQL

The SQL language has a few components:

- **Data Definition Language**: The SQL DDL provides commands for defining and deleting of relations schemas and modifying relation schemas.
- **Data Manipulation Language**: The SQL DML provides ability to query and insert information into relations.
- **Integrity**: DDL includes commands for specifying integrity constraints. The database is prohibited to enter a invalid state.
- **View definition**: DDL includes commands for defining views.
- **Transaction control**: SQL includes commands for specifying the beginning and ending of atomic transactions.
- **Embedded SQL** and **Dynamic SQL**: Defines how SQL statements can be embedded within general purpose programming languages such as C or Java.
- **Authorisation/Data Control Language**: DDL includes commands for specifying access rights.

SQL is based on relational calculus but has non-first order features. SQL is standardised by a committee.

3.1 Data Definition Language

SQL provides a rich DDL that allows one to define tables and constraints. For instance, to define the `department` table,

```
create table department
  ( deptname char(20)
  , building char(15)
  , funding numeric(12,2)
  , primary key(deptname)
  );
```

SQL has several basic data types:

- `char(n)`/`character(n)`: A fixed-length character string with specified length n.

  Spaces are automatically appended for shorter strings.
- `varchar(n)`/`character varying(n)`: Variable-length string with maximum length n.

  Using this type is recommended.
- `nvarchar(n)`: Unicode variant of `varchar`. On many platforms `varchar` can already store UTF-8 unicode.

- `int`/`integer`: An integer whose range is platform dependent.
- `smallint`: An integer whose range is platform dependent.
- `numeric(p,d)`: Fixed-point number with a sign, p digits, and that d of the p digits are fractional.
  e.g. `numeric(3,1)` can precisely store 23.1 but not 23.18 or 123.1.
- `real`, `double precision`: Floating point numbers.
- `float(n)`: Floating point number with at least n digits.

The special value `null` indicates an absence of value. In certain cases we may prohibit `null` values from being entered.

3.1.1 Schema Definition

Syntax

`create table` is the command to create a new relation. The general syntax for `create table` is

```
create table r
  ( A1 D1
  , A2 D2
  , ...
  , An Dn
  , integrity-constraint
  , ...
  , integrity-constraint
  );
```

where r is the name of the relation, each A_i is an attribute of type D_i, and the list of attributes is followed by integrity constraints.

The semicolon shown at the end of this and other SQL statements is optional in many implementations.

Syntax

`not null`: This constraint when specified on an attribute specifies that `null` is not allowed for this attribute.

3.1.2 Primary key and Foreign key

Syntax

`primary key`$(b_1, \ldots, b_m)$: An integrity constraint specifying that the attributes $b_1, \ldots, b_m$ form the primary key for the relation. The primary key attributes must be *non-null* and *unique*.

This is optional but it is generally a good idea to specify a primary key for each relation.

Syntax

`foreign key`$(b_1, \ldots, b_m)$`references` $s(c_1, \ldots, c_k)$: Specifies that $b_1, \ldots, b_m$ must correspond values to primary keys of some tuples in s.

`foreign key` clause can also have two actions:

```
foreign key (<attrs>)
  references <ref-table> (<attrs>)
  on delete <delete-action>
  on update <update-action>
```

where the actions can be

- `restrict`: Produce an error
- `cascade`: Propagate the delete. This cannot be applied on update.
- `set null`: Set to "unknown.

3.1.3 Check constraint

Syntax

`check <condition>`: Check for additional integrity constraints using a *simple* search condition.

`check` cannot be a subquery.

e.g.

```
create table EMP
  ( ssn         integer not NULL ,
    name        char(20),
    dept        integer,
    salary      dec(8,2),
    primary key (ssn),
    foreign key (dept) references Dept(id)
      on delete cascade
      on update restrict,
    check (salary > 0)
  )
```

The standard allows for *interrlational constraint* in `check` clause (anything that can go into `where` can go into `check`), but the DB2 implementation does not. A workaround is ECA triggers.

3.2 Data Manipulation Language

The SQL query language is non-procedural. A **query** takes as input several tables (usually only one) and always returns a single table. For example, the following command [1] returns all instructors in Statistics:

```
select distinct instructor.name
from instructor
where instruction.deptname = 'Statistics';
```

This query specifies that

- The `name` attribute of the matching rows are displayed.
- The `deptname` attribute of matching rows must be 'Statistics'.

[1] `distinct` will be discussed later

This query returns the table

name
Wu, Changbao
Hofert, Marius

Queries may involve information from more than one table. For instance, the following query finds the instructor `id` and department name of all instructors associated with a department with more funding than 200:

```
select distinct instructor.id, department.deptname
from instructor, department
where instructor.deptname = department.deptname and
      department.budget > 200;
```

id	deptname
3245	Physics
3291	Physics

3.2.1 Select Statement

Syntax

The `select` instruction has the format

```
select distinct <results>
from   <tables>
where  <condition>
```

which is equivalent to the Relational Calculus query

$$\left\{\text{results} : \exists \text{unused}. \bigwedge \text{tables} \wedge \text{condition}\right\}$$

The unused are variables not used in results.

For example:

```
select distinct *
from instructor;
```

returns all (distinct!) entries in the `instructor` table.

- Since many database in practice has relations of high arity, the *position* notation in relational calculus (e.g. $R(x, y, z)$) becomes inconvenient. SQL uses *co-relations* and *attribute names* to assign default variable names. $R\,[\texttt{AS}]p$ in SQL is $R(p.a_1, \ldots, p.a_n)$ in Relational Calculus, where $a_1, \ldots, a_n$ are attribute names declared in R.

  For example, to list publications with at least two authors:

  $$\{p : \exists a_1, a_2\, \mathsf{wrote}(a_1, p) \wedge \mathsf{wrote}(a_2, p) \wedge (a_1 \neq a_2)\}$$

  This can be done by

  ```
  select distinct r1.publication
  from wrote r1, wrote r2
  where r1.publication = r2.publication
  and r1.author != r2.author;
  ```

  We *cannot share* a variable p in the two wrote relations, so we force this by an explicit `r1.publication = r2.publication`

- Relations can serve as their own co-relations when unambiguous.

```
select distinct title
from publication, book
where publication.pubid = book.pubid;
```

1. `publication` is an abbreviation of
2. `publication publication` is an abbreviation of
3. `publication(publication.pubid, publication.title)`

Syntax

The `from` clause has the syntax

`from` R_1 `[[as]]` n_1`]` ,..., R_k `[[as]]` n_k`]`

`as` is optional since historically some queries don't have it.

Syntax

The `select` clause has the syntax

`select distinct`
a_1 `[[as]]` n_1`]` ,..., a_k `[[as]]` n_k`]`

`select` is allowed to form expressions (built-in functions applied to values of attributes). Built-in functions include:

- On numeric types: $+, -, \cdot, /$
- On strings: || (concatenation), substr
- Constants: e.g. `select 1`
- UDF (user defined functions)

Example:

```
select distinct pubid, endpage-startpage+1
from article;
```

This generates a table with columns pubid and endpage − startpage + 1. If we wish to use the query result somewhere else, we need to rename:

```
select distinct pubid                as id,
                endpage-startpage+1  as numberofpages
from article;
```

3.2.2 Where clause

Syntax

The `where` clause has the syntax

`where <condition>`

Find all journals printed since 1997:

```
select distinct *
from journal where year >= 1997;
```

Find all articles with more than 20 pages:

```
select distinct *
from article
where endpage - startpage > 20;
```

List all publications with at least two authors:

```
select distinct r1.publication
from wrote r1, wrote r2
where r1.publication = r2.publication
and not r1.author = r2.author;
```

Where subqueries:

- Additional (complex) search conditions
- Advantages: Simplifies writing queries with negation
- Drawbacks: Complicated semantics. Very easy to make mistakes.

3.2.3 Set operations

SQL cannot express something such as "list all publication titles's thats either a book or a journal". The or clause needs to be expressed as a set operation. Why is this not a solution?

```
select distinct title
from publication, book, journal
where publication.pubid = book.pubid
  or publication.pubid = journal.pubid
```

This *often* works, but has 2 problems:

- It generates a list of pairs (b, j) such that for each valid b j iterates through the set of all possible journals, and for each valid j, b iterates through all possible books. This is inefficient.
- If there are no books, or no journals, the `from` clause will abort and no tuples will be returned.

what would happen if there are no `books`?

Syntax

The **set operations** in SQL have the syntax:

- Union: `Q1 union Q2`
- Difference: `Q1 except Q2`
- Intersection: `Q1 intersect Q2`

Operands in a set operation must have **union-compatible** signatures.

If the two operands to a set operation have different columns, we must set columns to `null`:

```
select col1, col2, col3 from table1
union
select col1, col2, null as col3 from table2
```

Note. If we have $\mathrm{FV}(\varphi_2) \subseteq \mathrm{FV}(\varphi_1)$, the difference

$$\varphi_1 \wedge \neg\varphi_2$$

needs to be represented as

$$\varphi_1 \wedge \neg(\varphi_1 \wedge \varphi_2)$$

since $\mathrm{FV}(\varphi_1) = \mathrm{FV}(\varphi_1 \wedge \varphi_2)$. This is highly unpleasant to do since

If we want to use a set operation *inside* a `select` block, we can use distributive laws:

$$(A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$$

Unfortunately this is also very unpleasant since C is duplicated.

Syntax

SQL provides a subquery naming mechanism

```
with f1 [<opt-schema-1>] as ( query1 ),
     ...
     fn [<opt-schema-n>] as ( queryn )
<query that uses f1 ... fn as tables>
```

Example: "List all publication titles for books or journals":

```
with bookorjournal (pubid) as
    ( (select distinct pubid from book)
      union
      (select distinct pubid from journal)
    )
select distinct title
from publication, bookorjournal
where publication.pubid = bookorjournal.pubid;
```

If the query is only used once, there is one more shorthand: We can inline queries in the `from` clause.

```
select distinct title
from publication,
    ( (select distinct pubid from journal)
      union
      (select distinct pubid from book) ) as jb
where publication.pubid = jb.pubid;
```

`where` subquery allows for:

- Presence/Absence of a single value in a query:

  ```
  <attr> in ( <query> )
  <attr> not in ( <query> )
  ```

  Example:

  ```
  select distinct title
  from publication
  where pubid in (select pubid from article)
  ```

- Relationship of a value to some/all values in a query:

  ```
  <attr> <op> some ( <query> )
  <attr> <op> all ( <query> )
  ```

Example:

```
select distinct pubid
from article
where endpage-startpage >= all (
  select endpage-startpage
  from article
);
```

- Emptiness/Non-emptiness:

```
exists ( <query> )
not exists ( <query> )
```

3.2.4 Parametric Subqueries

So far subqueries were **independent** of the main query. SQL allows for **parametric** (correlated) subqueries. Parametric subqueries contain attributes in te main query. The *truth* of a predicate defined by a subquery is determined for each substitution (tuple) in the main query:

- Instantiate all the parameters used.
- Check for the truth value as before.

Example: Publications of at least two authors.

```
select *
from wrote r
where exists (
    select *
    from wrote s
    where r.publication = s.publication
    and r.author <> s.author
);
```

It is easy to now complement conditions

```
select *
from wrote r
where not exists (
    select *
    from wrote s
    where r.publication = s.publication
    and r.author <> s.author
);
```

`where` subqueries are just queries. We can nest again and again (although some architectures limit the layers of nesting). Every nested subquery acn use attributes.

This can be used to formula very complex search conditions. Attributes present in the subquery only *cannot* be used to construct the results.

"List of all authors who always publish with someone else"

$$\{n : \exists a_1, a_2, w.\mathsf{author}(a_1, n) \wedge \mathsf{author}(a_2, n) \wedge a_1 \neq a_2 \\ \wedge \forall p, t.\mathsf{publication}(p, t) \wedge \mathsf{wrote}(a_1, p) \rightarrow \mathsf{wrote}(a_2, p)\}$$

Since there are $\forall$ statements we need to convert it to positive form

- (Select 1) $\exists a_1, a_2, w.\mathsf{author}(a_1, n) \wedge \mathsf{author}(a_2, n) \wedge a_1 \neq a_2$
- (Select 2) $\neg\exists p, t.\mathsf{publication}(p, t) \wedge \mathsf{wrote}(a_1, p)$

- (Where 2) $\wedge\neg$wrote

```
select distinct a1.name
from author as a1, author as a2
where not exists (
  select *
  from publication as p, wrote as w1
  where p.pubid = w1.publication
    and a1.aid = w1.author
    and a2.aid not in (
      select author
      from wrote
      where publication = p.pubid
      and author <> a1.aid  -- a1 <> a2
    )
)
```

3.3 Insertion and Deletion

How do we modify a database? We could do:

```
dbstart;
r1 = Q1(DB1);
     ...
rn = Qn(DBn);
dbcommit;
```

This is not an acceptable solution in practice.

The idea of insertion and deletion is that tables are large but **updated are small**. 3 operations are available:

1. Insertion of a tuples (`insert`)
   - Constant tuple
   - Results of queries
2. Deletion of tuples (`delete`)
   - Based on match of a condition
3. Modification of tuples (`update`)
   - Allows updating in place
   - Based on match of a condition

Syntax

The `insert` statement allows insertion of a single tuple:

```
insert into r[(a1,...,ak)]
       values (v1,...,vk)
```

adds tuples $(v_1, \ldots, v_k)$ to r. The type of $(v_1, \ldots, v_k)$ must match the schema definition of r.

Or multiple tuples such as the result of a query

```
insert into r (Q)
```

Example:

```
insert into instructor
  values (3300, 'Tuncel', 'Mathematics', 60000);
```

Syntax

To delete with a condition:

```
delete from r
where <cond>
```

This deletes *all* tuples that match `<cond>`. If the condition is omitted, the entire table is wiped.

Syntax

An `update` statement has a `set` (update) part and a `where` (search) part.

```
update r
set <statement>
where <condition>
```

Example:

```
update author
set url = 'bricks.dk/~david'
where aid in (
    select aid
    from author
    where name like 'Toman%'
  )
```

3.4 Transactions

Syntax

To signal the end of a transaction, use

```
commit
```

There is no special command to signal the beginning of a transaction. A transaction starts with the first access of the database.

The commit can fail because

- If the modification breaks integrity constraints.
- If there is no sufficient authorisation to access the database.

A successful transaction emits the message:

```
SQL> commit;
Commit complete.
```

Syntax

A transaction can be reverted using the `rollback` command.

A successful rollback emits the message:

```
SQL> rollback;
Rollback complete.
```

3.5 Database Access

SQL is not Turing complete. SQL also does not support most I/O operations and network communication. These functionalities are written in a *host* language, such as C or Java with embedded SQL queries that access the database.

SQL is case insensitive.

Definition

A **view** is a relation whose instance is determined by the instances of other relations.

Types of views:

- **Virtual**: Views are used only for querying and not stored in database.
- **Materialised**: The query that makes the view is executed. The view constructed and stored in the database.

A view has many of the same properties as a table.

- Its schema information appears in the database schema.
- Access controls can be applied
- Other views can be defined in terms of a view.

Syntax

To define a view,

```
create view <name>
  as (<query>)
```

Example:

```
create view ManufacturingProjects
  ( select projno, projname, firstname, lastname
    from project, employee
    where respemp = empno and deptno = 'D21' )
```

Modifications to a view's instance must be propagated back to instances of relations in conceptual schema. Unfortunately, some views *cannot* be updated unambiguously.

In SQL-92, a view is **updatable** only if its definition satisfies

- The query references exactly one table.
- The query only outputs simple attributes (no expression)
- There is no grouping/aggregation/`distinct`
- There are no nested queries
- There are no set operations.

Problem.
When a base table changes, the materialised view may also change.

Solution:

- Periodically reconstruct the materialised view.
- Incrementally update the materialsed view

3.5.1 Data Control Language

Assigns access rights to database objects.

Syntax

```
grant  <what> on <object> to <user(s)>;
revoke <what> on <object> to <user(s)>;
```

where `<what> on <object>` may be

- `database`: `bindadd`, `connect`, `createtab`, `create_not_fenced`, `dbadm`
- `index`: `Control`
- `package`: `bind`, `control`, `execute`
- `table`/`view`: `alter`, `control`, `index`, `references`, `select`, `insert`, `delete`, `update`

`<user(s)>` is a list of

- `user <name>`
- `group <name>`
- `public`

In some implementations and configurations, the users may be synchronised to the UNIX user system. In some other configurations, the database defines its own users.

3.6 Limitation of First-Order SQL

SQL introduced so far captures all of Relational Calculus. It has several problems:

- Some queries are hard to write (syntactic sugar)
- No "counting" (aggregation)
- No "path in graph" (recursion)

Fortunately, SQL provides workarounds.

3.7 Aggregation

Syntax

The **aggregation** syntax in SQL uses the `select` keyword

```
select x1, ..., xk, agg1, ..., aggl
from Q
group by x1, ..., xk;
```

All attributes in the `select` clause that are not in the scope of an aggregate function *must* appear in the `group by` clause. `aggi` are of the form

- `count(*)`
- `count(<expr>)`
- `sum(<expr>)`
- `min(<expr>)`
- `max(<expr>)`
- `avg(<expr>)`

where `<expr>` is usually an attribute of `Q` and not in the `group by` clause.

Aggregation:

- Partition the input relation to groups with equal values of **grouping** attributes
- On each partition, apply the **aggreagate function**
- Collect the results to form the answer.

Example: For each publication, count the number of authors:

```
select publication, count(author)
from wrote
group by publication
```

For each author, count the number of article pages:

```
select author, sum(endpage-startpage+1) as pgs
from wrote, article
where publication=pubid
group by author;
```

This is not quite correct since if an author has never written any publications, the query does not give 0. Instead we need to separately query authors without a publication.

```
select author, sum(e-s+1) as pgs
from (select w, author, startpage as s, endpage as e
      from wrote as w, article as a
      where a.pubid = publication) t
group by author;
```

`where` clause cannot impose conditions on the value of aggregates. The `where` clause has to be used *before* `group by`. SQL allows a `having` clause, which is a syntactic sugar and can be replaced by a nested query and a `where` clause.

Example: List publications with exactly one author

```
select publication, count(author)
from wrote
group by publication
having count(author) = 1;
```

3.8 Order, Null, and Duplications

SQL has operations that are not parts of relational calculus.

3.8.1 Ordering Results

Syntax

In general form a query can be ordered by the clause

`order by` e_1 [Dir$_1$], ..., e_k [Dir$_k$]

The Dir$_i$ inputs are optional and can take one of two values of `asc` or `desc`.

3.8.2 Duplicate Semantics

SQL has a **multiset** semantics ather than a **set** semantics. SQL tables are multisets of tuples. The original motivation is for efficiency.

To allow for this, we need to extend the definition of range-restricted formula to use distinct.

Definition

If φ is range-restricted, then so is $\mathsf{distinct}(\varphi)$.

Idea:

- A **finite valuation** can appear $k > 0$ times as a query answer.
- The number of duplications is a *function* of the number of duplicates in formulae.
- $\mathbf{DB}, \theta, k \models \varphi$ reads "finite valuation θ appears k times in φ's answer.

Definition

Multiset Semantics for Relational Calculus:

$$
\begin{aligned}
\mathbf{DB}, \theta, k &\models R(x_1, \ldots, x_k) && \text{if } R \in \rho, (\theta(x_1), \ldots, \theta(x_k)) \in \mathsf{R} \text{ k times} \\
\mathbf{DB}, \theta, (m \cdot n) &\models \varphi \wedge \psi && \text{if } \mathbf{DB}, \theta, m \models \varphi \text{ and } \mathbf{DB}, \theta, n \models \psi \\
\mathbf{DB}, \theta, m &\models \varphi \wedge (x_i = x_j) && \text{if } \mathbf{DB}, \theta, m \models \varphi \text{ and } \theta(x_i) = \theta(x_j) \\
\mathbf{DB}, \theta, \sum_{v \in D} n_v &\models \exists x.\varphi && \text{if } \mathbf{DB}, \theta[x := v], n_v \models \varphi \\
\mathbf{DB}, \theta, (m + n) &\models \varphi \vee \psi && \text{if } \mathbf{DB}, \theta, m \models \varphi \text{ and } \mathbf{DB}, \theta, n \models \psi \\
\mathbf{DB}, \theta, (m - n) &\models \varphi \wedge \neg\psi && \text{if } \mathbf{DB}, \theta, m \models \varphi \text{ and } \mathbf{DB}, \theta, n \models \psi, (m > n) \\
\mathbf{DB}, \theta, 1 &\models \varphi && \text{if } \mathbf{DB}, \theta, m \models \varphi
\end{aligned}
$$

Under this new set of rules, the equality

$$(\underbrace{A}_{n} \wedge \underbrace{B}_{m}) \vee \underbrace{C}_{l} \equiv (A \vee C) \wedge (B \vee C)$$

The formula

$$(n \cdot m) + l \neq (n + l) \cdot (m + l)$$

Recall that the syntactic sugar

```sql
select r.b
from r
where r.a in (
  select b
  from s
)

select r.b
from r, (
  select distinct b
  from s
) as s
where r.a = s.b
```

Rewriting does not generally hold if `distinct` is removed.

Syntax

SQL has the following multiset operations. Let Q_1, Q_2 be two queries. Suppose a tuple $\boldsymbol{x}$ occurs in Q_1 m times and Q_2 n times, it will occur in the output $Q_1 \setminus Q_2$ $\max\{m-n, 0\}$ times.

- `union all`: Additive union multiset: $\boldsymbol{x}$ will occur in $Q_1 \cup Q_2$ $m+n$ times.
- `except all`: Subtractive difference: $\boldsymbol{x}$ will occur in $Q_1 \setminus Q_2$ $\max\{m-n, 0\}$ times.

  This operator does not remove duplicates.
- `intersect all`: Multiset of all tuples taking the maximum number common to Q_1 and Q_2. i.e. $\boldsymbol{x}$ will occur in $Q_1 \cap Q_2$ $\min\{m, n\}$ times.

3.8.3 Nulls

Avoid nulls.

`null` indicates an *absence of value*.

	phone	
Name	Office	Home
Joe	1234	3456
Sue	1235	?

In the table above, Sue might not have a home phone, or that Sue's home phone is unknown. This is essentially a poor schema design, but may be chosen for efficiency and reflects the physical format of data storage.

Workarounds:

- Store office and home phones separately.

office phone	
Name	Office
Joe	1234
Sue	1235

home phone	
Name	Home
Joe	3456

- Fill in the unknown value with a placeholder such as 0000.

How do we answer queries? We can try to answer common to all possible worlds:

$$Q(D) \coloneqq \bigcap_{W \text{ world of } D} Q(W)$$

This is computationally infeasible (NP-hard to undecidable).

SQL provides a crude approximation:

- Expressions: General rule is that a `null` in an expression should make the result `null`.
- Predicates/Comparisons: Three-valued logic.
- Set operations: Unique special values for duplicates
- Aggregate operations: Does not contribute to the aggregate total.

Comparison with a `null` returns `unknown`. Unfortunately, this produces an extra number of unknowns in logic expressions. e.g.

$$x = 0 \vee x \neq 0$$

with $x =$ `unknown` will return `unknown`.

`unknown` can be used in a where clause:

- `is true`, `is false`, `is unknown`.

  Note: `where <cond>` is shorthand for `where <cond> is true`.
- Special comparison `is null`.

List all authors for which we don't know a URL of their home page.

```
select aid, name
from author
where url is null;
```

`count (URL)` counts only non-`null` URL's. The special `count (*)` operation counts the number of tuples including `null`'s, e.g.

```
select count) as rs, count(url) as usfrom author;
```

Gives `rs` $= 3$, `us` $= 2$.

3.9 Joins

Syntax

The **join** allows `null`-padded answers that fail to satisfy a conjunct in a conjunction. The `join` syntax is an extension of the `from` clause:

`from` R `<j-type>` `join` S `on` C

The `<j-type>` is one of `full`, `left`, `right`, `inner`.

The below table summarises the semantics (for $R(x,y), S(y,z), C = (r.y = s.y)$):

$\{(x,y,z) : \dots\}$	`full`	`left`	`right`	`inner`
$(R(x,y) \wedge S(y,z))$	✓	✓	✓	✓
$\vee((z = \mathsf{null}) \wedge R(x,y) \wedge \neg(\exists z.S(y,z)))$	✓	✓		
$\vee((x = \mathsf{null}) \wedge S(y,z) \wedge \neg(\exists x.R(x,y)))$	✓		✓	

3.10 Other SQL Features

3.10.1 Recursive Query

The below query finds transitive closure

```
with tc as (select e.x, t.y from edge as e, tc as t,
    where e.y = t.x)
    union
    (select * from edge)
select * from tc
```

This is a valid query, but how can we understand this?

$$\mathsf{tc}(x, y) \leftarrow (\mathsf{edge}(x, y) \vee (\exists z.\mathsf{edge} \wedge \mathsf{tc}(z, y)))$$

SQL is still not Turing complete with this, since the query will always terminate. But adding arithmetic operations to recursive queries is Turing complete.

3.11 Embedded SQL

SQL is not sufficient to write general applications. Generally another application written in another language need to interface between the user and the database server.

3.11.1 C-SQL Interface

SQL statements are *embedded* into a *host* language. This means the application is written in a mixture of SQL and C, and then *pre-compiled* to be pure C code.

- Advantages: Preprocessing static parts of queries becomes possible. Easy to use
- Disadvantages: Needs precompiler. Needs to be bound to a database.

Application structure:

1. Include SQL support (SQLCA, SQLDA)

   Include SQL communication area: (This is C code!)

   ```
   EXEC SQL INCLUDE SQLCA;
   ```

2. Declarations

   SQL statements can have parameters that are host variables in the embedding language. Host variables communicate *single values* and must be declared within SQL declare sections

   ```
   EXEC SQL BEGIN DECLARE SECTION;
     char db[6] = "DBCLASS";
   EXEC SQL END DECLARE SECTION;
   ```

   Host variables can be used in `EXEC SQL` statements using a colon to distinguish them from SQL identifiers.

3. Connect to Database

4. Compute

   SQL statements are inserted using magic words

   ```
   EXEC SQL <statement>;
   ```

5. Process Errors: We could check `sqlcode != 0`, or use "exception handling":

   ```
   EXEC SQL WHENEVER SQLERROR   GO TO label;
   EXEC SQL WHENEVER SQLWARNING GO TO label;
   EXEC SQL WHENEVER NOT FOUND  GO TO label;
   ```

   `label` has to be in scope.

6. Commit/Abort and Disconnect.

Example

```
#include <stdio.h>
#include "util.h"

EXEC SQL INCLUDE SQLCA;

int main(int argc, char *argv[])
{
  EXEC SQL BEGIN DECLARE SECTION;
    char db[6] = "DBCLASS";
  EXEC SQL END DECLARE SECTION;
  printf("Sample␣C␣program\n");
  EXEC SQL WHENEVER SQLERROR GO TO error;
  EXEC SQL CONNECT TO :db;
  printf("Connected␣to␣DB2\n");
  // Compute
  EXEC SQL COMMIT;
  EXEC SQL CONNECT reset;
  exit(0);
error:
  check_error("My␣error", &sqlca);
  EXEC SQL WHENEVER SQLERROR CONTINUE;
  EXEC SQL ROLLBACK;
  EXEC SQL CONNECT reset;
  exit(1);
}
```

To compile a program like this,

1. Store the application in a file `<name>.sqc`
2. Preprocess: `db2 prep <name>.sqc`
3. Compile: `cc -c -O <name>.c`
4. Link (with DB2 libraries) `cc -o <name> <name.o> -L... -l...`
5. Execute `./<name>`

3.11.2 Null and Indicator variables

To pass in a `null` value, we can use a **indicator**:

```
smallint ind;
select firstname into :firstname
                  indicator :ind
from ...
```

If `ind < 0` then `firstname` is `null`. If this indicator was not provided and null generates, we get a *run-time error*. The same rules apply for host variables in updates.

3.11.3 Impedance Mismatch

What if `EXEC SQL` returns more than one tuple?

1. Declare the *cursor*:

   ```
   EXEC SQL DECLARE <name> CURSOR
            FOR <query>;
   ```

2. Iterate on cursor

```
EXEC SQL OPEN <name>;
EXEC SQL WHENEVER NOT FOUND GO TO end;
while (SQLCODE == 0) {
  /* set up host parameters */
  EXEC SQL FETCH <name>
           INTO <host-variables>;
  /* process fetched tuple */
}
end:
EXEC SQL CLOSE <name>;
```

⚠. DB2 has a bug for which only the first of the host variables in the `INTO` clause can be prepended by a colon:

```
EXEC SQL FETCH author INTO :name, title;
```

If there was a colon before `title`, the code will not compile.

3.11.4 Stored Procedures

A stored procedure executes application logic directly inside the DBMS process. This has several advantages:

1. Minimise data transfer costs
2. Centralise application code
3. Logical independence

Example:

```
create function sumSalaries(dept CHAR(3)) returns decimal(9,2)
language sql
return
  select sum(salary)
  from employee
  where workdept = dept
```

Usage:

```
select deptno, sumSalaries(deptno) as sal
from department
```

3.11.5 Dynamic SQL

Goal: Execute a *string* as a SQL statement. Problems:

- How do we know a string is a valid and safe statement?
- How do we execute queries and updates?
- What if we don't know anything about the string?

(Dynamic SQL Roadmap)

Syntax

Execution of **non-parametric** statements **without answers**:

```
EXEC SQL execute immediate :string;
```

where `:string` is a host variable containing the ASCII representation of the query, which may not return an answer or contain parameters. `:string` is compiled every time we pass through.

Caput 4

Entity-Relation Modeling

Used for (and designed for) database (conceptual schema) design.

In designing a database schema, we avoid two categories of problems:

- Redudancy: A bad design may repeat information.
- Incompleteness: A bad design may make certain aspects of the enterprise difficult or impossible to model.

4.1 Entities, Attributes, Relations

The *world* of E-R modeling are described in terms of

- Entities
- Attributes
- Relationships

Definition

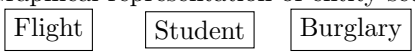
A **entity** is a distinguishable object. A **entity set** is a set of entities of the same type.

Entities are represented as rectangles.

A **strong entity set**'s elements can be identified uniquely by its attributes. A **weak entity set**'s elements cannot.

Examples of entity sets: Student enrolled, Flights offered by NASA, burglaries in Ontario during 1994.

Graphical representation of entity sets

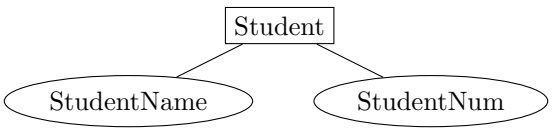


Definition

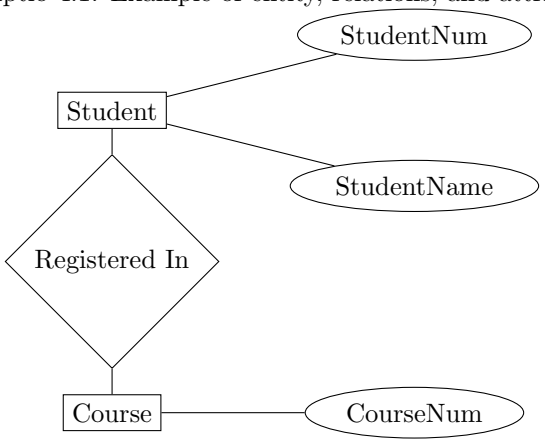
An **attribute** describes properties of entities. The **domain** is the set of permitted values for an attribute.

Attributes are represented as ellipse.

Each entity has a **value** for its attributes.



Descriptio 4.1: Example of entity, relations, and attributes



Definition

A **relationship** is a representation of the fact that certain entities are related to each other.
A **relationship set** is a set of relationships of a given type.
Relationships are represented as rhombuses.

If $E_1, \ldots, E_n$ are entity sets, a relationship set R is a subset of

$$\{(e_1, \ldots, e_n) : e_i \in E_i\}$$

and each element in R is a relationship.

Examples: Students registered in courses, Passengers booked on flights.

Definition

The association between entity sets is the **participartion**. That is, $E_1, \ldots, E_n$ **participate** in the relationship R. The participation is **total** if every entity in E participates in at least one relationship in R. Otherwise the participation is **partial**.
Total participation is indicated by a double line =.

A **relationship instance** in an E-R schema represents an association between the named entities that is being modeled. e.g. An instructor and a student may participate in a relationship instance of *advisor*. An relationship may also have attributes. For example, the *advisor* relationship may contain a *date* attribute.

Relationships may be *recursive*, such as one which models the dependencies between courses.

The graphical representation of E-R model is equivalent to formulae such as

- $\forall x, y\, \mathsf{RegisteredIn}(x, y) \to (\mathsf{student}(x) \wedge \mathsf{course}(y))$
- $\forall x\, \mathsf{course}(x) \to \exists \mathsf{cnum}(x, y) \wedge \mathsf{Int}(y)$
- $\forall x, y\, \mathsf{cnum}(x, y) \wedge \mathsf{cnum}(x, z) \to y = z$

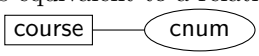
There could be multiple relationships between entities and attributes.

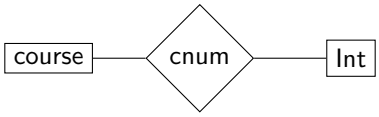
Definition

A **role** is the function of an entity set in a relation set.
A **role name** is an explicit indication of a role.

Role labels are needed when an entity set has multiple functions in a relationship set.

Descriptio 4.2: An integer attribute is equivalent to a relation between an entity set and the set of integers





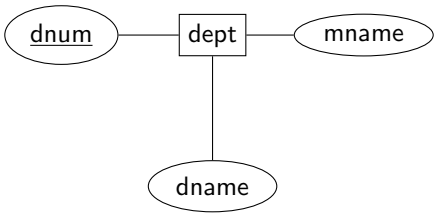
4.2 Constraints in E-R Models

Each entity must be distinguishable from any other entity in an entity/set by its attributes.

Definition

A **primary key** is a selection of attributes chosen by designer values of which determines a particular entity.

Primary keys are labeled with underlines.



If the relationship set R has no attributes associated with it, the set of attributes

$$\mathsf{primarykey}(E_1) \cup \cdots \cup \mathsf{primarykey}(E_n)$$

descibes an individual relationship in R. If R ahs attributes $a_1, \ldots, a_m$ associated with it, the set

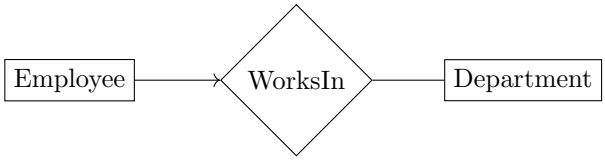
$$\mathsf{primarykey}(E_1) \cup \cdots \cup \mathsf{primarykey}(E_n) \cup \{a_1, \ldots, a_m\}$$

describes an individual relationship in set R.

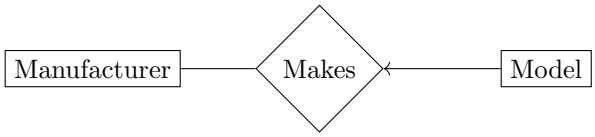
Types of relations:

- Many-to-many: An entity in one set can be related to many entities in the other set, and vice versa.

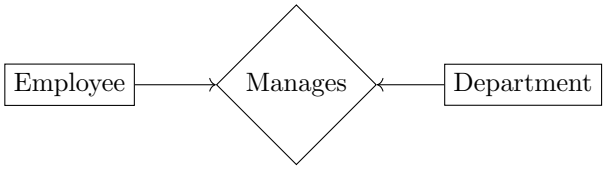
- Many-to-one: Each entity in one set can be related to at most one entity in the other, but not vice versa.



- One-to-many: Similar,



- One-to-one: Each entity in one set can be related to at most one entity in the other.



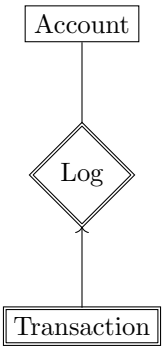
The idea of "one" is represented by an arrow.
Sometimes the existence of an entity depends on the existence of another entity.

Definition

If x is **existence dependent** on y, then

- y is a **dominant entity**
- x is a **subordinate entity**

Example: Transactions existence dependent on accounts. Notice moreover that this is a many-to-one relationship.



Consider a *section* entity, which is uniquely identified by a course identifier (cid), semester (s), year (y), and section identifier (sid). Suppose we create a relationship set (sectioning) between entity sets section and course. Then the cid in section is redundant, since the section is already related to a course. One possible solution is to remove the relationship sectioning, but then the relationship between section and course becomes implicit in an attribute, which is bad practice.

An alternative way is to eliminate the cid attribute. However, the entity section then does not ahve enough attributes to identify a particular section entity uniquely, but the unique identification is reliant on the fact that section is subordinate to course.

Definition

A **weak entity set** is an entity set containing subordinate entities. The entity set which the aforementioned set is subordinate to, is the **identifying** or **owner** entity set. The relationship associating the weak entity set and the identifying entity set is the **identifying relationship**.

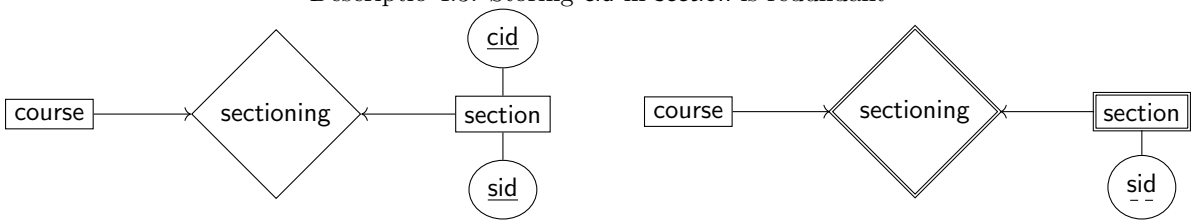
The **discriminator** of a weak entity set is the set of attributes that distinguish subordinate entities of the set w.r.t. a particular dominant entity.

A **strong entity set** is an entity set containing no subordinate entities.

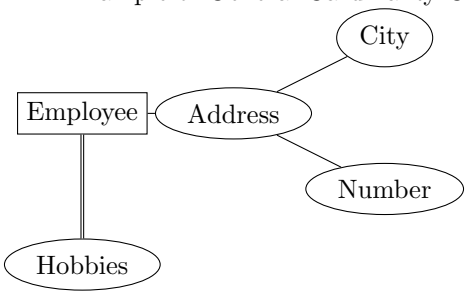
The identifying relationship is signified with double line.

A weak entity set must have a many-to-one relationship to a distinct entity set.

Descriptio 4.3: Storing cid in section is redundant

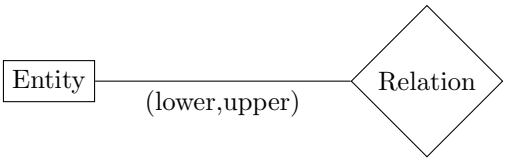


Descriptio 4.4: Example of General Cardinality Constraints



Definition

General Cardinality Constraints determine lower and upper bounds on the number of relationships of a given relationship set in which a component entity may participate.



4.3 Extensions to E-R Modeling

4.3.1 Structured Attributes

An attribute, as used in E-R model, can be characterised by types:

- **Simple** vs. **Composite**: So far the attributes are *atomic*, or simple. Simple attributes have no constituent parts. Composite attributes, on the other hand, can be divided into subparts.

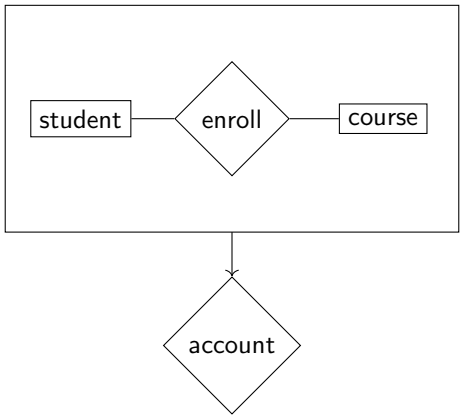
  Using composite attributes in a design schema is a good choice if a user will refer to an entire attribute on some occassions. e.g. telephone numbers.

- **Single-valued** and **Multi-valued** attributes. A common example of a multi-valued attribute is phone number. A person may have $0, 1, 2$, or more phone numbers. When appropriate, upper and lower bounds may be placed on the number of values in a multivalued attribute.

- **Derived** attribute: The value for this type of attribute can be derived from the values of other related attributes. For example, *age* can be derived from *dateOfBirth*.

4.3.2 Aggregation

Relationships can be viewed as higher-level entities. For example:

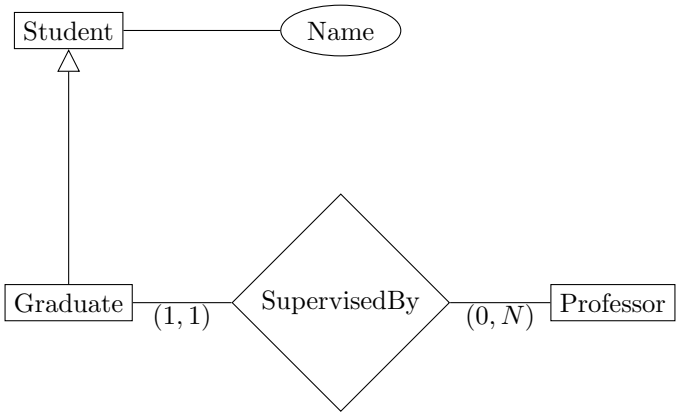


i.e. A course account is allocated for a student enrollment in the course.

4.3.3 Specialisation

Definition

A **specialisation** is a specialised kind of entity set which may be derived from a given entity set.



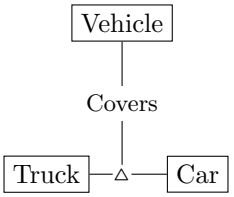
4.3.4 Generalisation

Definition

In **generalisation**, several entity sets can be abstracted by a more general entity set.

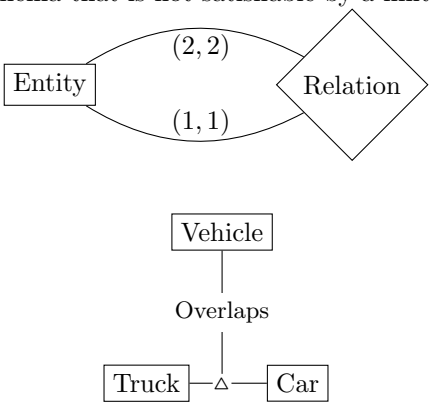
In **total generalisation**, each higher-level entity must belong to a lower-level entity set. Otherwise the generalisation is **partial**.

"A vehicle abstracts the notion of a car and truck"



Specialised entity sets are usually disjoint but can be declared to have entities in common. For example, we may decide that nothing can both be a car and a truck. However, we can declare them to overlap:

Descriptio 4.5: A schema that is not satisfiable by a finite number of entities

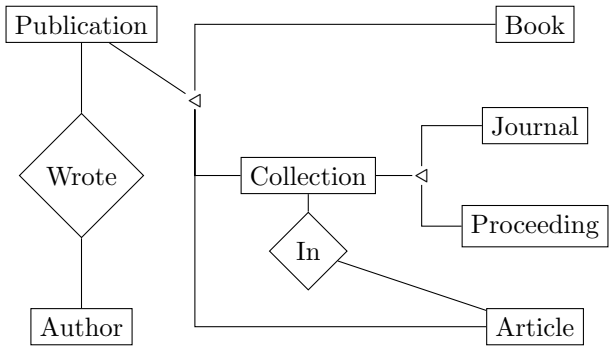


4.4 Designing an E-R Schema

Points to consider:

- Use attribute/entity set?
- Use entity set or relationship set?
- Degrees of relationships?

The publication database can be expressed as



Notice that the new entity set collection is required. Moreover, articles are no longer publication's pubid's.

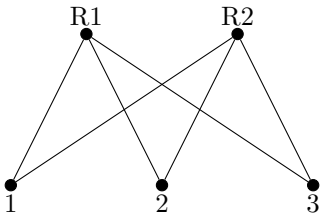
We can view entities, relations, and attributes as relations:

- Entity: $P(x)$
- Relation: $R(x_1, \ldots, x_n)$
- Attribute: $A(x, y)$

In some cases, data can be stored as either attributes or entity sets. Example: Should one model employees' phone numbers by a PhoneNumber attribute or by a Phone entity set related to the Employee entity set? Rule of thumb:

- Is it a separate object?
- Do we maintain information about it?
- Can several of its kind belong to a single entity?
- Does it make sense to delete such an object?

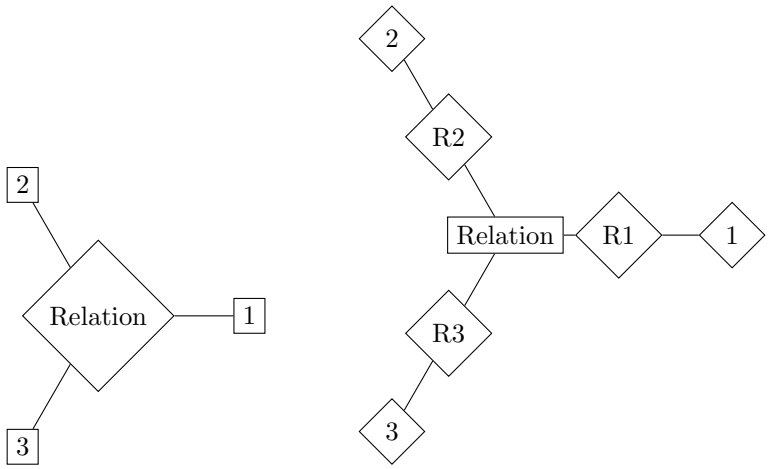
Descriptio 4.6: Multiple relation object corresponding to the same set of entities



- Can it be missing from some of the entity set's entities?
- Can it be shared by different entities?

An affirmative answer to any of the above suggests a new entity set.

Moreover, we can represent a relationship on n entity sets with n binary relationships.



This is subtly different from a single relationship since we can now have multiple "Relation" objects corresponding to one set of entities $1, 2, 3$. This is previously impossible since relations are sets.

4.5 Reduction to Relational Schemas

We can represent a database that conforms to an E-R database schema by a collection of relation schemas.

Main idea:

- Each entity set maps to a new table
- Each attribute maps to a new table column
- Each relationship set maps to either new table columns or a new table.
- Representing *strong* entity sets:

  E with attributes $a_1, \ldots, a_n$ translates to table E with attributes $a_1, \ldots, a_n$. Each entity is a row in the table and the primary keys are the primary keys of the table.
- Complex attributes are given separate tables (e.g. phone numbers)
- Representing *weak* entity sets:

  Let E be a weak entity set with attribwtes $a_1, \ldots, a_n$. Let F be the identifying entity set with primary key $b_1, \ldots, b_m$.

E can be expresented by a relation schema with columns

$$\{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_m\}$$

More generally, the table representation for E should include

- Attributes of the weak entity set
- Attributes of the identifying relationship set
- Primary key attributes of entity set for dominating entities.

- Let R be a relationship set. Let $a_1, \ldots, a_m$ be the set of attribtues formed by the union of primary keys of each entity sets participating in R. Let $b_1, \ldots, b_n$ be the descriptive attribtues. Then R is represented by the relation schema with columns

  $$\{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_m\}$$

  Primary key of table R determined as follows:

  - If we can deduce the general cardinality constraint $(0, 1)$ for a component entity set E, then take the primary key attributes for E.
  - Otherwise, the primary key is the union of primary key attributes of each component entity.

- If a relationship set is an identifying relationship set for a weak entity set then no action is needed.

- If we can deduce the general cardinality constraint $(1, 1)$ for a component entity set E then add following columns to table E:

  - Attributes of the relationship set
  - Primary key attribtues of remaining component entity sets.

- An aggregation is treated like an entity set whose primary key is the primary key of the relation in aggregation.

- Representing specialiseation:

  Create table for higher-level entity set and treat specialised entity subsets like weak entity sets (without discriminators)

- Representing generalisation has two methods:

  1. Create table for each lower-level entity set *only*. Columns of new tables should include attributes of superset and lower level entity set.

     The higher-level entity set can be defined as a view.

  2. Treat generalisation using the same method as specialisation.

Caput 5

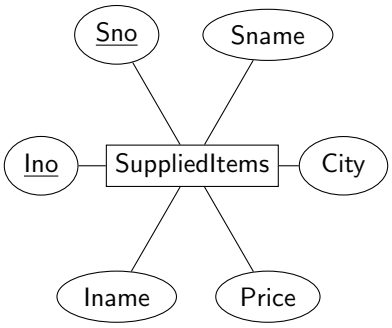
Normalisation Theory

When we get a relational schema:

- How do we know if it good?
- What to watch for?

5.1 Change Anomalies

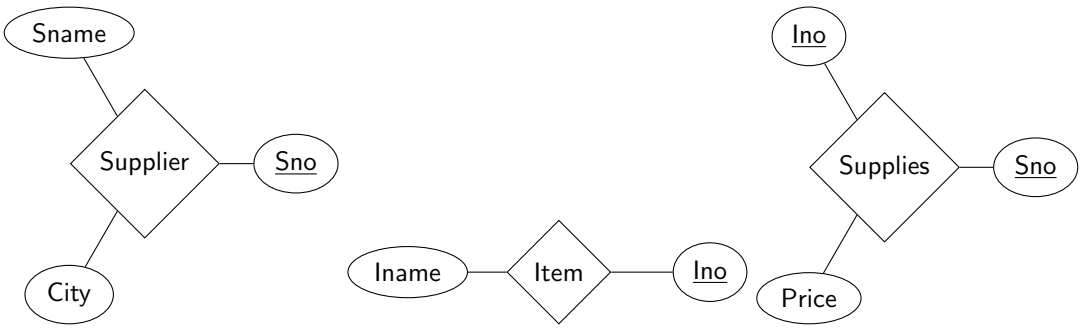
Assume we are given the E-R diagram



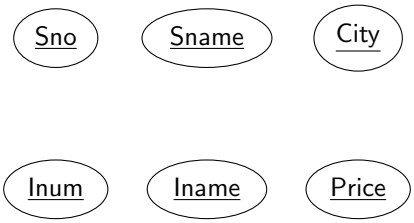
Problems: (**anomalies**)

- Update problem: (Changing the name of a supplier)
- Insert problems (New item w/o supplier)
- Delete prolems (Budd no longer supplies screws)
- Likely increase in space requirements

The table below is much better:



But the other extreme where we split the table as much as possible also has problems. Information about relationships can be lost.



How to find and fix anomalies?

- Detection: (certain families of) integrity constraints postulate regularities in schema instances that lead to anomalies.
- Repair: Certain schema decompositions avoid the anomalies.

5.2 Functional Dependencies

An instance of a relation that satisfies all real constraints is a **legal instance** of the realtion. A legal instance of a database is one where all the relation instances are legal instances.

To ease expression, we will use the following notation:

- Latin letters such as $r(R)$ represents a relation, and R its attributes.
- Greek letters refer to sets of attributes.

FDs express the fact that in a relation schema, a set of attributes uniquely determines another set of attributes.

Definition

Let R be a relation schema. Let $\alpha, \beta \subseteq R$ be sets of attributes. The **functional dependency** $\alpha \to \beta$ is the formula

$$\forall v_1, \ldots, v_k, w_1, \ldots, w_k . R(v_1, \ldots, v_k) \wedge R(w_1, \ldots, w_k) \wedge (\bigwedge_{j \in \alpha} v_j = w_j) \to (\bigwedge_{j \in \beta} v_j = w_j)$$

i.e.

$$\forall t_1, t_2 . t_1[\alpha] = t_2[\alpha] \to t_1[\beta] = t_2[\beta]$$

α **functionally determines** β. A trivial functional dependency is of the form $\alpha \to \beta \subseteq \alpha$.

$\alpha \to \beta$ **holds** on schema $r(R)$ if in every legal instance of $r(R)$ it satisfies the functional dependency.

Definition

Let $r(R)$ be a relation schema. A subset $K \subseteq R$ is a **superkey** if $K \to R$.

Examples: In the `EMP` table,

- `sin` determines employee name.

$$\texttt{sin} \to \texttt{name}$$

- Project number determines project name and location

$$\texttt{pname} \to \texttt{pname}, \texttt{ploc}$$

- Allowances are always the same for the same number of hours at the same location

$$\texttt{ploc}, \texttt{hours} \to \texttt{allowance}$$

Definition

A domain of attributes is **atomic** if elements of the domain are considered to be indivisible units. If the domains of all attribtues of a relation schema R, R is in **1st normal form**.

A list of telephone numbers is not atomic. Note that the use of course identifiers such as "CS100" may be atomic as long as the database application does not attempt to split the identifier and interpret its parts separately.

Consider the following relation schema:

SIN	PNum	Hours	EName	PName	PLoc	Allowance

The following functional dependencies hold:

- SIN determines employee name: SIN $\rightarrow$ EName
- Project number determines project name and location: PNum $\rightarrow$ PName, PLoc
- Allowances are always the same for the same number of hours at the same location: PLoc, Hours $\rightarrow$ Allowance.

Definition

$K \subseteq R$ is a **candidate key** for relation schema R if K is a superkey and no subset of K is a superkey.

A primary key is a candidate key chosen by the DBA.

Below is an example of a table with $2^{|R|/2}$ candidate keys.

$$R = (\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n)$$
$$\mathcal{F} := \{\alpha_i \rightarrow \beta_i, \beta_i \rightarrow \alpha_i : 1 \leq i \leq n\}$$

A candidate key of R w.r.t. $\mathcal{F}$ contains exactly one of $\{A_i, B_i\}$ for each i, and therefore has 2^n candidate keys.

5.2.1 Boyce-Codd Normal Form

Definition

Schema R is in **Boyce-Codd Normal Form** (**BCNF**) w.r.t. a set of FD's $\mathcal{F}$ if: whenever $(\alpha \rightarrow \beta) \in \mathcal{F}^+$ and $\alpha, \beta \subseteq R$, then either

- $\alpha \rightarrow \beta$ is trivial (i.e. $\beta \subseteq \alpha$)
- α is a superkey of R.

A database schema $\{R_1, \ldots, R_n\}$ is in BCNF if each relation schema R_i is in BCNF.

Rule of thumb:

- Independent facts in separate tables.
- Each relation schema should consist of a prime key and a set of mutually exclusive attributes.

Example of a non-BCNF table InstDept:

id	name	salary	deptname	building	budget

This table combines the instructors and departments into one. The functional dependency deptname $\rightarrow$ budget holds on InstDept, but deptname is not a superkey. The decomposition of this table into instructor and department is a better design.

We now state a general rule for decomposing non-BCNF schema: Let R be a non-BCNF scehma. Then there exists a non-trivial functional dependency $\alpha \rightarrow \beta$. We repalce R in the design with two schemata:

- $\alpha \cup \beta$
- $R - (\beta - \alpha)$

This leads to the following algorithm which computes a BCNF for schema R w.r.t. $\mathcal{F}$:

```
1: procedure ComputeBCNF(R, F)
2:     D := {R}
3:     while ∃R_i ∈ D, (α → β) ∈ F⁺ : violate BCNF condition
4:         D.replace(R_i, R_i − (β − α))
5:         D.add(α ∪ β)
6:     end while
7:     return D
8: end procedure
```

Unfortunately, no efficient procedure to do this exists. Results depend on sequence of FD's used to decompose the relations.

How do we enforce constraints on the decomposed schema?

Consider the FD set

$$\mathcal{F} := \{\alpha \to \beta, \beta \to \gamma, \alpha \to \gamma\}$$

and decomposition

$$R(\alpha, \beta, \gamma) \equiv D_1 = \begin{cases} R_1(\alpha, \beta) \\ R_2(\beta, \gamma) \end{cases}$$

$$\equiv D_2 = \begin{cases} R_1(\alpha, \beta) \\ R_2(\alpha, \gamma) \end{cases}$$

Both are lossless. Which is better?

- D_1 lets us test $\alpha \to \beta$ on table R_1, and $\beta \to \gamma$ on table R_2. If they are both satisfied, then $\alpha \to \gamma$ is automatically satisfied.
- In D_2, $\beta \to \gamma$ is a **inter-relational constraint**: Testing requires joining tables R_1 and R_3.

Hence D_1 is better.

Definition

Let $D = \{R_1, \ldots, R_n\}$ be a decomposition of R.

A constraint $\alpha \to \beta \in \mathcal{F}$ is **inter-relational** if testing it requires joining two tables in D.

D is **dependency preserving** if there is an equivalent set $\mathcal{F}'$ of functional dependencies, for which is none is inter-relational in D.

5.3 Closure

How do we know what additional FD's hold in a schema?

Definition

Let $\mathcal{F}$ be a set of functional dependencies that hold on relation $r(R)$.

$\mathcal{F}$ **logically implies** a functional dependency $\alpha \to \beta$, if in *all* legal instances of $r(R)$ that satisfies $\mathcal{F}$, $\alpha \to \beta$ holds in $r(R)$.

The **closure** $\mathcal{F}^+$ of $\mathcal{F}$ is the set of all functional dependencies that are logically implied by $\mathcal{F}$. $\mathcal{F}, \mathcal{G}$ are **equivalent** if $\mathcal{F}^+ = \mathcal{G}^+$.

For example:

$$\mathcal{F} = \{\alpha \to \beta, \beta \to \gamma\}$$
$$\mathcal{F}^+ = \{\alpha \to \beta, \beta \to \gamma, \alpha \to \gamma\}$$

$\alpha\beta$ represents the concatenation of α and β.
Logical implications can be derived using

Armstrong's Axioms

- (Reflexivity) $\beta \subseteq \alpha \implies \alpha \to \beta$
- (Augmentation) $\alpha \to \beta \implies \alpha\gamma \to \beta\gamma$
- (Transitivity) $\alpha \to \beta, \beta \to \gamma \implies \alpha \to \gamma$

5.3.1 Proposition.
Let $\alpha, \beta, \gamma, \delta$ be sets of attribtues. Then

1. *(Union)* $\alpha \to \beta . \alpha \to \gamma \implies \alpha \to \beta\gamma$
2. *(Decomposition)* $\alpha \to \beta\gamma \implies \alpha \to \beta$
3. *(Pseudo-transitivity)* $\alpha \to \beta, \gamma\beta \to \delta \implies \alpha\gamma \to \delta$

Proof.

1. - (Augmentation) Since $\alpha \to \beta$ we have $\alpha \to \alpha\beta$.
   - (Augmentation) Since $\alpha \to \gamma$ we have $\alpha\beta \to \alpha\gamma\beta \to \beta\gamma$.
   - (Transitivity): $\alpha \to \alpha\beta \to \beta\gamma$
2. $\beta \subseteq \beta\gamma$ because of reflexivity. Using transitivity, $\alpha \to \beta$
3. Since $\alpha \to \beta$, by augmentation we have $\alpha\gamma \to \beta\gamma \to \delta$.

□

Below is an algorithm to determine if an FD is implied by $\mathcal{F}$. Given α, it computes α^+, the closure of α under $\mathcal{F}$.

```
1: procedure CLOSURE(α, F)
2:     α⁺ ← α
3:     while true
4:         if ∃(β → γ) ∈ F : β ⊆ α⁺, γ ⊈ α⁺
5:             α⁺ ← α⁺ ∪ γ
6:         else
7:             return exit
8:         end if
9:     end while
10:    return α⁺
11: end procedure
```

5.3.2 Theorem.
Let $r(R)$ be a relational schema and $\mathcal{F}$ a set of functional dependencies.

- *α is a superkey of R iff*
$$\text{CLOSURE}(\alpha, \mathcal{F}) = R$$
- *$\alpha \to \beta \in \mathcal{F}^+$ iff*
$$\beta \subseteq \text{CLOSURE}(\alpha, \mathcal{F})$$

5.3.1 3rd Normal Form

Definition

A relation schema R is in **3rd normal form**[a] (**3NF**) w.r.t. a set $\mathcal{F}$ of FD's, if for all for all functional dependencies $\alpha \to \beta \in \mathcal{F}$ with $\alpha, \beta \subseteq R$, at least one of the following holds:

- $\alpha \to \beta$ is trivial.
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R[b].

[a] We skipped 2nd normal form, since it is of historical significance only and is not used in practice.
[b] Does not have to be the same candidate key

3NF is a relaxation of BCNF. It is not obvious that the 3rd alternative in 3NF is useful. It represents a minimal relaxation of BCNF conditions that ensures every schema has a dependency-preserving decomposition into 3NF.

Definition

A set of dependencies $\mathcal{F}$ is **minimal** if

- Every right-hand side of an dependency in $\mathcal{F}$ is a single attribute.
- For no $\alpha \to \gamma$ is the set $\mathcal{F} \setminus \{\alpha \to \gamma\}$ equivalent to $\mathcal{F}$. i.e. No dependency can be removed.
- For no $\alpha \to \gamma$ and $\beta \subset \alpha$ we have $\mathcal{F} \setminus \{\alpha \to \gamma\} \cup \{\beta \to \gamma\}$ equivalent to $\mathcal{F}$. i.e. The left side of any dependency in $\mathcal{F}$ is unshrinkable.

5.3.3 Theorem.
*For every set of dependencies $\mathcal{F}$ there is an equivalent minimal set of dependencies (**minimal cover**).*

How to find a minimal cover? We repeat each step until it no longer succeeds in updating $\mathcal{F}$.

1. Replace $\alpha \to \beta\gamma$ with $\{\alpha \to \beta, \alpha \to \gamma\}$
2. Remove $\alpha \to \gamma$ from $\mathcal{F}$ if $\gamma \in \textsc{Closure}(\alpha, \mathcal{F} - \{\alpha \to \gamma\})$
3. Remove β from the left side of $\alpha \to \gamma$ in $\mathcal{F}$ if $\gamma \in \textsc{Closure}(\alpha - \{\gamma\}, \mathcal{F})$
   (we have a minimal cover here)
4. (Merge): Replace $\alpha \to \beta, \alpha \to \gamma$ in $\mathcal{F}$ with $\alpha \to \beta\gamma$.

Unlike BCNF, 3NF can be efficiently computed.

```
1:  procedure COMPUTE3NF(R, F)
2:      D := {}
3:      F' ← MINIMAL-COVER(F)
4:      for (α → β) ∈ F'
5:          D ← D ∪ {αβ}
6:      end for
7:      if ¬∃R_i ∈ D : ∃K.K ⊆ R_i, K is candidate key for R
8:          K ← CANDIDATE-KEY(R)
9:          D ← D ∪ {K}
10:     end if
11:     return D
12: end procedure
```

5.4 Lossless-Join Decomposition

Definition

Let R be a relation schema (set of attributes). The set $\{R_1, \ldots, R_n\}$ of relation schemata is a **decomposition** of R is

$$R = R_1 \cup \cdots \cup R_n$$

A good decomposition does not:

- Lose information
- Complicate checking of constraints
- Contain anomalies (or at least contains fewer anomalies)

Definition

A decomposition $\{R_1, R_2\}$ of R is **lossless** if and only if the common attributes of R_1, R_2 form a superkey for either schema,

$$R_1 \cap R_2 \to R_1, R_2$$

$\{R_1, R_2\}$ is a **lossless-join decomposition**.
A non-lossless decomposition is **lossy**.

ComputeBCNF computes a lossless-join BCNF decomposition. Unfortunately, it is possibel that no lossless-join dependency preserving BCNF decomposition exists. e.g.

$$R = \{\alpha, \beta, \gamma\}, \qquad \mathcal{F} = \{\alpha\beta \to \gamma, \gamma \to \beta\}$$

i.e. relation r contains the same set of tuples as the result of the following SQL query:

```
select *
from (select R1 from r)
    natural join
    (select R2 from r)
```

i.e.

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

5.4.1 Theorem.
Let R_1, R_2 be a decomposition of R and $\mathcal{F}$ a set of functional dependencies. $\{R_1, R_2\}$ is a lossless decomposition of R if at least one of the following functional dependencies is in $\mathcal{F}^+$:

- $R_1 \cap R_2 \to R_1$
- $R_1 \cap R_2 \to R_2$

i.e. $R_1 \cap R_2$ forms a superkey of either R_1 or R_2.

To digest this, we consider the schema

$$\mathsf{InstDept}(\mathsf{id}, \mathsf{name}, \mathsf{salary}, \mathsf{deptname}, \mathsf{building}, \mathsf{budget})$$

which we decomposed into two schemata:

$$\begin{cases} \mathsf{instructor}(\mathsf{id}, \mathsf{name}, \mathsf{salary}, \mathsf{deptname}) \\ \mathsf{department}(\mathsf{deptname}, \mathsf{building}, \mathsf{budget}) \end{cases}$$

The intersection of these two schemata is deptname, which is a superkey for department.

5.4.1 Beyond Functional Dependencies

There are anomalies/redundancies in relational schemata that cannot be captured by FD's. e.g. A teacher may decide to use multiple textbooks in a course: (This is the Course-Teacher-Book schema)

course	teacher	book
Math	Smith	Algebra
Math	Smith	Calculus
Math	Jones	Algebra
Math	Jones	Calculus
Pure Math	Smith	Calculus
Physics	Black	Mechanics
Physics	Black	Optics

There are no non-trivial FD's that hold on this schema.

5.5 Multivalued Dependencies

Functional dependencies rule out certain tuples from being in a relation. If $\alpha \to \beta$, we cannot have two tuples with the same α but different β values. Multivalued dependencies, do not rule out the existence of certain tuples. Instead they require that other tuples of a certain form to be present. Hence functional dependencies are sometimes referred to as **equality-generating dependencies** and multivalued dependencies are referred to as **tuple-generating dependencies**.

Definition

Let $r(R)$ be a relational schema and $\alpha, \beta \subseteq R$. The **multivalued dependency** $\alpha \twoheadrightarrow \beta$ holds if for any legal instance of relation $r(R)$,f or all parits of tuples $t_1, t_2 \in r$ such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples $t_3, t_4 \in r$ such that

$$\begin{cases} t_1[\alpha] = t_2[\alpha] & = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] & = t_1[\beta] \\ t_4[\beta] & = t_2[\beta] \\ t_3[R-\beta] & = t_2[R-\beta] \\ t_4[R-\beta] & = t_1[R-\beta] \end{cases}$$

Note. This means if we "cross" t_1, t_2, we get t_3, t_4.

Definition

A **dependency basis** for α with respect to a set of FD's and MVD's $\mathcal{F}$ is a partition of $R - \alpha$ into sets $\beta_1, \ldots, \beta_k$ such that $\mathcal{F} \models \alpha \twoheadrightarrow \gamma$ iff $\gamma - \alpha$ is a union of some of the Y_i's.

- Unlike for FD's, we cannot split right-hand sides of MVD's to single attributes.
- The dependency basis for α w.r.t. $\mathcal{F}$ can be computed in polynomial time.

Similar to the FD case, we want to decompose the schema to avoid anomalies.

5.5.1 4th Normal Form

Definition

Let R be a relational schema and $\mathcal{F}$ a set of FD's and MVD's. R is in 4NF if and only if whenever $\alpha \twoheadrightarrow \beta \in \mathcal{F}^+$ and $\alpha\beta \subseteq R$, then either

- $\alpha \twoheadrightarrow \beta$ is trivial ($\beta \subseteq \alpha$ or $\alpha\beta = R$)
- α is a superkey of R

A database schema $\{R_1, \ldots, R_n\}$ is in 4NF if each relation schema R_i is in 4NF.

The Course-Teacher-Book schema can be decomposed into two schemata.

[Decomposition]

5.6 Other dependencies

Definition

A **Join dependency** on R, denoted $\bowtie [R_1, \ldots, R_k]$ holds if

$$\forall \boldsymbol{x}.R(x) \leftrightarrow (R_1(\boldsymbol{x}_1) \wedge \cdots \wedge R_k(\boldsymbol{x}_k))$$

where $\forall \boldsymbol{x}_i.R_i(\boldsymbol{x}_i) \iff \exists.R(\boldsymbol{x}_i, \boldsymbol{y})$ holds for all $0 < i \leq k$.

This is a generalisation of an MVD. $\alpha \twoheadrightarrow \beta$ is the same as $\bowtie [\alpha\beta, X(R - \beta)]$. This **cannot** be simulated by MVD's.

No axiomatisation exists for join dependencies.

Strength of normal forms:

$$\text{4NF} \leq \text{BCNF} \leq \text{3NF}$$

Caput 6

Database Implementations

6.1 Query Execution

Applications:

- User interaction: Query input, presentation of results
- Application specific tasks

Database Server:

- DDL evaluation
- DML compilation selection of a query plan for a query
- DML execution
- Concurrency control
- Buffer Management

File system:

- Storage and retrieval of unstructured data

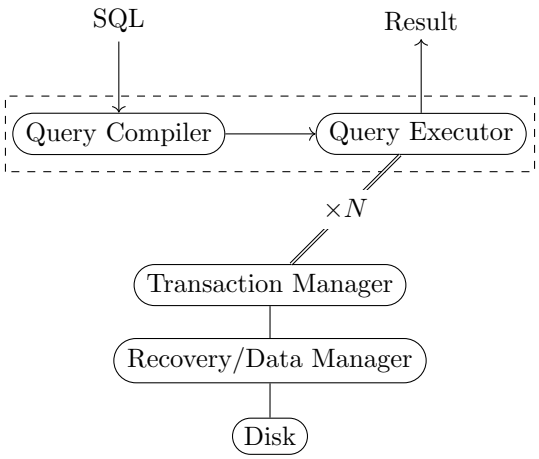
A **relational calculator** that answers queries have the following considerations:

- How is data physically represented?
- How to compute answers to complex queries?
- How are intermediate results managed?

To implement the query language, we use Codd's Theorem and convert a query to a relational algebra expression. Then we can implement every operator naïvely. For example, below is an implementation of $\sigma_{\cdot_i=\cdot_j}$ (select):

```
OPERATOR selection(OPERATOR c, int i, int j)
  : child_(c), i_(i), j_(j) {}
void open() { child_.open(); }
void close() { child_.close(); }
tuple fetch()
{
    tuple t = child_->fetch();
    if (t == NULL || t.attr(i_) == t.attr(j_))
      return t;
    return this->fetch();
}
```

Descriptio 6.1: Execution of A query. Transaction Manager is responsible for consistency and integrity. The recovery/data manager is responsible for atomicity.



The rest of the physical operators can be implemented with:

- Product: Nested loops
- Projection: Eliminate unwanted attributes
- Union: Concatenation
- Difference: Nested loops

Also, projection and union need to be followed by a **duplication elimination** operator.

This implementation is *very (very very)* slow.

6.2 Query Optimisation

There are some ways to speed up queries:

- Use (disk-based) data structures for efficient searching.
- Use better algorithms to implement the operators, commonly based on *sorting* or *hashing*.
- Rewrite the RA expression to an equivalent but more efficient one. Remove unnecessary operations.

6.2.1 Atomic Relations and Indexing

WWhen an index $R_i(x)$ (where x is the search attribute) is available, we replace a subquery of the form $\sigma_{x=c}(R)$ with accessing $R_i(x)$ directly. Even if an index is available, scanning the entire relation may be faster in certain circumstances. e.g. When the relation is very small, or when we expect most tuples in the relation to satisfy the selection criteria.

An index on attribute A of a realtion is **clustering** if tuples in the relation with similar values of A are stored together. Other indices are **non-clustering**. A relation may have at most one clustering index.

There are many **query plans** for a single query:

- Equivalences in relational algebra
- Choice of operator implementation.

Finding the optimal plan is undecidable (consider a query which has no answers). Even for *conjunctive* queries, the To find a approximately best plan,

- "Always good" transformations

- Cost-based model

The performance of database operations depend on the way queries and updates are executed w.r.t. a particular *physical schema design*.

6.2.2 Cost Evaluation

A simple cost model for disk I/O has the following assumptions:

- Uniformity: All possible values of an attribute are equally likely to appear.
- Independence: The likelihood that an attribute has a particular value (in a tuple) does not depend on values of other attributes.

For a stored relation with an attribute A, we keep:

1. $|R|$, cardinality.
2. $b(R)$: Blocking factor (how many entries are stored in a disk block)
3. $\min(R, A)$: Minimum value of A in R
4. $\max(R, A)$: Maximum value of A in R.
5. $\mathsf{distinct}(R, A)$: Number of distinct values.

The cost of an operation using an index A is:

$$\mathsf{cost} \coloneqq \begin{cases} \delta + N/b(R) & \text{if } A \text{ is clustered} \\ \delta + N & \text{if } A \text{ is not clustered} \\ |R|/b(R) & \text{if scanning entire relation} \end{cases}$$

δ is the depth of the B-tree.

The cost of **physical** operations, in number of I/Os:

- Selection: $\mathsf{cost}(\sigma_c(E)) = (1 + \epsilon_c)\mathsf{cost}(E)$
- Nested-loop join: (R is the **outer** relation)

$$\mathsf{cost}(R \bowtie S) = \mathsf{cost}(R) + (|R|/b)\mathsf{cost}(S)$$

- Index join:

$$\mathsf{cost}(R \bowtie S) = \mathsf{cost}(R) + \delta\,|R|$$

- Sort-Merge join

$$\mathsf{cost}(R \bowtie S) = \mathsf{cost}(\mathsf{sort}(R)) + \mathsf{cost}(\mathsf{sort}(S))$$

where

$$\mathsf{cost}(\mathsf{sort}(R)) = \mathsf{cost}(R) + \frac{|R|}{b}\log\frac{|R|}{b}$$

Why don't we always use sort-merge join? The sort step has some overhaead.

In the cost estimation we need to know sizes of results of operations.

Definition

The **selectivity** for a condition $\sigma_\varphi(r)$ is

$$\text{sel}\,\sigma_\varphi(r) \coloneqq \frac{|\sigma_\varphi(r)|}{|r|}$$

The optimiser will *estimate* selectivity with statistical rules

$$\begin{aligned}
\operatorname{sel} \sigma_{A=c}(r) &\simeq \frac{1}{\mathsf{distinct}(r, A)} \\
\operatorname{sel} \sigma_{A \leq c}(r) &\simeq \frac{c - \min(r, A)}{\max(r, A) - \min(r, A)} \\
\operatorname{sel} \sigma_{A \geq c}(r) &\simeq \frac{\max(r, A) - c}{\max(r, A) - \min(r, A)}
\end{aligned}$$

The estimates for joins are:

- General join:
$$|r \bowtie s| \simeq |r| \frac{|s|}{\mathsf{distinct}(s, A)} \simeq |s| \frac{|r|}{\mathsf{distinct}(r, A)}$$
- Foreign key join:
$$|r \bowtie s| = |s| \frac{|r|}{|s|} = |r|$$

So far, these are very primitive cost estimates. In partice we have more:

- Histogram to approximate non-uniform distributions.
- Correlations between attributes
- Uniqueness (keys) and containment (inclusions)
- Sampling methods
- Others

6.2.3 Always-Good Transformations

These heuristics work for majority of the cases.

- Push selections:
$$\sigma_\varphi(e_1 \bowtie_\theta e_2) = \sigma_\varphi(e_1) \bowtie_\theta e_2$$
if φ involves E_1 only.
- Push projections:
$$\Pi_V(r \bowtie_\theta s) = \Pi_V(\Pi_{V \cap R}(r) \bowtie \Pi_{V \cap S}(s))$$
- Replace products by joins:
$$\sigma_\varphi(r \times s) = r \bowtie_\varphi s$$

Joins are associative. A join $r \bowtie s \bowtie t$ can be written as $(r \bowtie s) \bowtie t$ or $r \bowtie (s \bowtie t)$. We minimise the intermediate results.

6.3 Pipelined and Parallel Plans

All operators (except sorting) operate without storing intermediate results. Iterator protocols are in constant storage. A **left-deep** plan does not require recomputation.

We introduce an additional operator **store**, which allows us to store intermediate results in a relation. Semantically, this operator represents identity.

The cost of store is

- Cumulative: (Compute value and store them in a relation)
$$\mathsf{cost}_c(\mathsf{store}(e)) = \mathsf{cost}_c(e) + \mathsf{cost}_s(e) + \frac{|e|}{b}$$

- Scanning: (Read all tuples in the stored result)

$$\mathsf{cost}_s(\mathsf{store}(e)) = \frac{|e|}{b}$$

Another approach to improving performance is to take advantage of hardware parallelism. Relational operators can be amended to have parallel execution.

6.4 Transaction Management

Query and update processing converts requests for *sets of tuples* to requests for reads and writes for physical objects. Database objects can be

- Individual attributes
- Records
- Physical pages
- Files

The goal of transaction processing:

- Correct and concurrent execution of queries and updates
- Ensure persistency and acknowledgement of updates

Transaction need to satisfy the **ACID properties**:

- Atomicity: All-or-nothing execution
- Consistency: Execution preserves database integrity
- Isolation: Transactions execute independently
- Durability: Updates made by a committed transaction will not be lost by subsequent failures.

The implementation of transactions in a DBMS comes in two parts:

- Concurrency control: Transactions do not interfere
- Recovery Management: Committed transactions are durable. Aborted transactions have no effect.

6.4.1 Transaction Scheduling

For performance reasons we cannot have a global lock on the database and allow transaction through one-by-one, but it must appears that this is the case. We have the following assumptions:

1. We fix a database: A set of objects read/written by transactions.
   - $r_i[x]$: Transaction T_i reads object x
   - $w_i[x]$: Transaction T_i writes object x
2. A transaction T_i is a sequence of operations:

   $$T_i \coloneqq r_i[x_1], r_i[x_2], w_i[x_1], \ldots, r_i[x_4], w_i[x_2], c_i$$

   c_i is the **commit request** of T_i.
3. For a **set of transactions** $T_1, \ldots, T_k$, we want to produce a *schedule* S of operations such that
   - Every operation $o_i \in T_I$ appears in S

- T_i's operations in S are ordered the same way as in T_i.

The goal is to produce a *correct schedule* with maximal parallelism.

If T_i, T_j are concurrent transactions, it is always correct to schedule the operations in such a way that

- T_i will appear to precede T_j. i.e. T_j will see all updates made by T_i and not vice versa.
- T_i will appear to succeed T_j.

Definition

An execution of S is **serialisable** if it is equivalent to a serial execution of the same transaction.

We can use equivalence to transform schedules. To determine if two schedules are equivalent:

- Conflict Equivalence: Two operations *conflict* if they
  1. Belong to different transactions
  2. Access the same data item x
  3. At least one of them is a write operation $w[x]$

  In two **conflict-equivalent histories** all conflicting operations are ordered the same way.
  A schedule is **conflict-serialisable** if it is conflict-equivalent to a serial schedule.
- View Equivalence: Allows more schedules, but is harder (NP-hard) to compute.

6.5 Recovery

Serialisability ensures correctness. Consider the following situation for transactions T_1, T_2:

1. T_1 writes a value.
2. T_2 reads a value T_1 has written.
3. T_2 succeeds to commit.
4. T_1 tries to abort.

To abort T_1 we need to undo effects of T_2.

Definition

A **recoverable schedule** commits only in the order of read-form dependency.
A **cascadeless schedule** (ACA) does not have reading of uncommitted data, i.e.

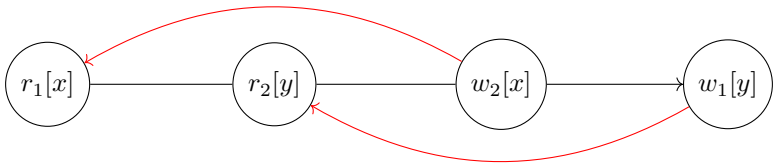
We wish to build schedulers that produce serialisable and cascadeless schedules.

Definition

The **schedule** receives requests from query processors. For each operation it chooses one of the following actions:

- Execute it (send to a lower module)
- Delay it
- Reject it (causing transaction to abort)
- Ignore it

Descriptio 6.2: Deadlock in two transactions



There are two kinds of schedulers, *conservative* (favours delaying operations) and *aggressive* (favours rejecting operations).

Transactions must have a *lock* on objects before access.

- A **shared lock** is required to read an object
- A **exclusive lock** is required to write an object.

It is insufficient to just acquire a lock, access the item, and then release immediately.
In a **2PL Protocol**, a transaction has to acquire all of the locks before it releases any of them.

6.5.1 Theorem.
Two-phase locking ensures that the produced transaction schedules are conflict serialisable.

In practice we use **strict 2PL**, i.e. locks are held until commit, ensuring ACA. With 2PL we may end with a *deadlock*: We can *prevent* and *detect* deadlocks.

- Locks are granted only if they cannot lead to a deadlock
- Order data and items and ensure that the locks can only be acquired in this order.
- Graph cycle detection
- Abort offending transactions if it causes a deadlock.

There are also variants on locking:

- Multi-granularity Locking: Not all locked objects have the same size
- Predicate locking: Locks based on a selection predicate rather than a value.
- Tree locking: Tries to avoid congestion in roots of B-trees. This allows some relaxation of 2PL due to the tree structure of data.
- Lock Upgrade Protocols

The two transactions below interfere in DB2 (i.e. the result will be a mix of 0's an 1's), because the `v=0` clause only locks the rows for which $v = 0$. However, if v is a primary key, then the transaction works.

```
update R set v = 1 where v = 0
update R set v = 0 where v = 1
```

Two implementations of recovery:

1. Shadowing:
   - Copy-on-write and merge-on-commit
   - Poor clustering
   - Used in system R but not modern systems
2. Logging
   - Use of LOG (separate disks) to avoid forced rewrites
   - Good utilisation buffers
   - Preserves original clusters

6.5.1 Log-Based Methods

[Log based methods]

If the power goes down during the middle of a transaction, there may be problems.

6.6 Physical Database Design

Physical design is the process of selecting a physical schema (collection of data structures) to implement the conceptual schema.

Tuning is the periodic adjusting of the physical and/or conceptual schema of a working system to adapt to changing requirements and/or performance characteristics.

Good design and tuning requires understanding the database workload.

Definition

A **workload description** contains:

- Most important queries and their frequency
- Most important updates and their frequency
- The performance goal of each query/update.

6.6.1 Creating Indexes

A table scan such as

```
select *
from employee
where lastname = 'Smith'
```

needs to visit every block of the file if the table is not sorted w.r.t. last name.

Syntax

The `create index` command

- Reduces the execution time for selections with conditions involving `key`.
- Increase execution time for insertions
- Increase/decrease execution time for updates or deletions of tuples from `employee`.
- Increase amount of space required to represent `employee`.

```
create index <Index-Name>
on <table>(<key>) [CLUSTER]

drop index <Index-Name>
```

e.g.

```
create index lastnameIndex
on employee(lastname) [CLUSTER]
```

6.6.2 More Complex Designs

- Multi-attribute indices
- Join Indices
- Materialised views

The more complex the design, the harder it is for query optimiser to use it.

6.6.3 Schema Tuning and Normal Forms

Vertical Partitioning

6.7 Recent Developments

From a user's point of view, the goal of a DBMS is to execute user queries/updates as fast as possible. Typicla requirements:

1. Stores all of you data (scalability)
2. Physical data independence (SQL vs. B-trees)
3. Atomicity/Durability (Transaction)
4. Isolation (Sharing/Concurrency)

Not all of the above are needed all the time.
Scale up:

- Parallel Databases (DB2/PE): Focus on query processing and algorithms
- Distributed Databases: Focus on transaction processing and ACID.

Partitioning and/or Replication

6.7.1 Evolution of DBMS

Before Y2K, DBMS is dominated by the big four (Oracle, IBM/DB2, Sybase, MS Server)
Today there are dozens and dozens of DBMS, because of two changes:

1. Cheap, abundant hardware
2. Changes in applications/workloads (big data)
3. Cost (commodity vs enterprise)

Standard Architecture: Client-Server System:

- Query/Update compiler
- Query/Update Execution engine

[Cost Estimation]

6.7.2 NoSQL Systems

NoSQL systems mean **not relational** database systems (not just SQL).

Advantages of NoSQL

- No/Flexible Schema
- Simple Interface (OO: No impedance mismatch)
- Horizontal Scale Out (on cheap hardware): Easy to add new machines to the cluster.
- Low (no) Admin overhead

Typical representatives:

- Key-value store (BerkeleyDB, DynamoDB)
  Pretty much B+tree
- Document store, "Key-JSON/XML store": (MongoDB)
  Allows data structures and queries on JSON. Each key's value is a JSON/XML
- Wide-Column store, "Key-multiple values store" (BigTable)
  Only table scans/selections. Complex queries in host language
- Graph DBMS: (Neo4J)
  Essenentially relational
  Certain relations (node, edges) are preferred. Allows for non first-order queries such as paths in graphs.

There is no such thing as "no schema". This usually means

- Schema is implicit in the data
- Schema is part of data (RDFS)

6.7.1 CAP Theorem.

You can only simultaneously ensure two of the following three properties:

1. *Consistency*
2. *Availability*
3. *Partition tolerance*

1. Consistency/Availability: 2 Phase commit, or Cache validation
2. Consistency/Partitioning: Locking and Majority Protocols
3. Availability/Partitioning: Leases and Conflict Resolution (Hard)

6.7.3 ACID vs. Eventual Consistency (BASE)

Eventual Consistency is BASE because acid is opposite to base.

- Data values may diverage but will eventually converge to the same value when no more updates are injected
- Trades consistency for availability and performance.

Session consistency:

- Data operation is consistent per session (user session)
- Allows asynchronicity of update propagation.

6.8 Big Data Characteristics

6.8.1 MapReduce

MapReduce is an architecture for data analysis of very large datasets. Data are stored as (key, value) pairs. MapReduce executes two functions, map and reduce:

- $\mathsf{map}(k_1, v_1) = \mathsf{list}(k_2, v_2)$
- $\mathsf{reduce}(k_2, \mathsf{list}(v_2)) = \mathsf{list}(v_3)$

Example:

```
f_map :: [Word] -> [(Word, Int)]
f_map li = map (\x -> (x, length x)) li

f_reduce :: Word -> [Int] -> (Word, Int)
f_reduce word counts = (word, (sum counts))
```

MapReduce does not work well in *iterative* computations.