

Informational Systems

December 10, 2025

Molnar Botond

Contents

| | |
|--|-----------|
| 1 Data Models and its Implementations | 5 |
| 1.1 Conceptual Data Model | 5 |
| 1.1.1 Characteristics | 5 |
| 1.2 Representational Data Model | 5 |
| 1.2.1 Characteristics | 6 |
| 1.3 Physical Data Model | 6 |
| 1.3.1 Characteristics | 6 |
| 2 Relational Model | 6 |
| 2.1 Definition of the Relation | 8 |
| 2.1.1 The Other Definition | 8 |
| 2.2 The NULL value | 8 |
| 3 Integrity Constraints | 9 |
| 3.1 Domain Constraint | 9 |
| 3.2 Key Constrains | 9 |
| 3.2.1 Superkey | 9 |
| 3.2.2 Key | 9 |
| 3.3 Entity Integrity Constarint (Egyedintegritasi megszoritas) | 10 |
| 3.4 Referential Integrity Constraint (Hivatkozasintegritasi megszoritas) | 10 |
| 3.5 Other Type of Constraints | 10 |
| 4 Relational Database and Relational Database Schema | 10 |
| 4.1 Operations on a Relation | 11 |
| 4.2 ACID Properties | 11 |
| 5 Object Relational Model | 11 |
| 5.1 Weaknesses of RDBMS | 11 |
| 5.2 ORDBMS | 12 |
| 5.3 User Defined Type (UDT) | 12 |
| 5.3.1 Distinct Type | 12 |
| 5.4 Structured UDT | 13 |
| 5.4.1 Viselkedés és szemantika | 13 |

| | | |
|-----------|---|-----------|
| 5.4.2 | Öröklődés | 13 |
| 5.4.3 | Observer és Mutator metódusok | 13 |
| 5.4.4 | Konstruktorok | 14 |
| 6 | XML Databases | 14 |
| 6.1 | Key Characteristics | 14 |
| 6.2 | Data-Centric vs. Document-Centric | 14 |
| 6.3 | Database Architectures | 14 |
| 6.3.1 | 1. XML-Enabled Databases (XEDB) | 14 |
| 6.3.2 | 2. Native XML Databases (NXD) | 15 |
| 6.4 | Query Languages | 15 |
| 6.4.1 | XPath (XML Path Language) | 15 |
| 6.4.2 | XQuery | 16 |
| 6.5 | Summary comparison | 16 |
| 7 | NoSQL Databases | 16 |
| 7.1 | Aggregátum adatmodellek | 17 |
| 7.1.1 | Előnyei | 19 |
| 7.1.2 | Hátrányai | 19 |
| 7.2 | Kulcs- érték és dokumentum adatmodellek | 19 |
| 7.3 | Oszlopcsalád tárak | 20 |
| 7.4 | Gráf adatbázisok | 21 |
| 7.5 | Sémamentes adatbázisok | 22 |
| 7.6 | Elosztási modellek | 23 |
| 7.6.1 | Sharding | 23 |
| 7.6.2 | Master-Slave replikáció | 24 |
| 7.6.3 | Peer-to-Peer replikáció | 25 |
| 7.6.4 | A sharding és a replikáció kombinációja | 25 |
| 7.6.5 | Konziszencia | 25 |
| 8 | ER Model | 28 |
| 8.1 | Egyed | 29 |
| 8.2 | Attribútumok | 29 |
| 8.2.1 | Egyedtípus | 29 |
| 8.2.2 | Azonosító | 29 |
| 8.2.3 | Értékhalmaz (vagy értéktartomány) | 29 |
| 8.2.4 | Kapcsolat | 29 |
| 8.2.5 | Bináris kapcsolattípusok strukturális megszorításai | 30 |
| 8.3 | Gyenge egyedtípusok | 30 |
| 9 | ER to Relational | 31 |
| 10 | EER Model | 33 |
| 10.1 | Alosztály, szuperosztály, és öröklődés | 33 |
| 10.1.1 | Típus öröklődés | 33 |
| 10.2 | Specializáció | 34 |
| 10.2.1 | Generalizáció | 34 |
| 10.2.2 | A specializáció és generalizáció megszorításai | 35 |
| 10.2.3 | Uniótípus modellezése kategóriák használatával | 35 |

| | |
|--|-----------|
| 11 EER modell leképezése relációs modellre | 36 |
| 12 Normal Forms | 37 |
| 12.1 A funkcionális függés definíciója | 37 |
| 12.2 A funkcionális függés tulajdonságai | 38 |
| 12.3 Lezárt | 38 |
| 12.4 Armstrong axióma | 38 |
| 12.5 Első Normálforma | 38 |
| 12.6 Részleges függés | 39 |
| 12.7 Második Normálforma | 39 |
| 12.8 Tranzitív függés | 39 |
| 12.9 Harmadik Normálforma | 39 |
| 12.10 Boyce-Codd Normálforma | 40 |
| 12.11 Többértékű függés | 40 |
| 12.12 Negyedik normálforma (4NF) | 41 |
| 13 UML | 41 |
| 13.1 Interaction models | 41 |
| 13.1.1 Use case modeling | 41 |
| 13.1.2 Sequence diagrams | 42 |
| 13.1.3 Class Diagrams | 43 |
| 13.1.4 Behavioral models | 47 |
| 14 Query Optimization | 48 |
| 14.1 SQL Processing | 48 |
| 14.1.1 SQL Parsing | 48 |
| 14.1.2 SQL Optimization | 49 |
| 14.1.3 SQL Row Source Generation | 49 |
| 14.1.4 SQL Execution | 50 |
| 14.2 Query Optimizer Concepts | 50 |
| 14.2.1 Cost-Based Optimization | 51 |
| 14.2.2 Execution Plans | 51 |
| 14.3 Optimizer Components | 52 |
| 14.3.1 Query Transformer | 52 |
| 14.3.2 Query Transformer | 52 |
| 14.3.3 Estimator | 53 |
| 14.3.4 Plan Generator | 53 |
| 14.4 Query Transformations | 54 |
| 14.4.1 OR Expansion | 54 |
| 14.4.2 View Merging | 54 |
| 14.4.3 Predicate Pusing | 56 |
| 14.4.4 Join Factorization | 56 |
| 14.5 SQL Operators: Access Paths and Joins | 57 |
| 14.5.1 Table Access Paths | 57 |
| 14.6 Optimizer Controls | 60 |
| 15 Software development | 60 |
| 15.1 Software process models | 61 |
| 15.1.1 The waterfall model | 61 |

| | |
|--|-----------|
| 15.1.2 Incremental development | 62 |
| 15.1.3 Reuse-oriented software engineering | 63 |
| 15.2 Process activities | 64 |
| 16 Architectural Design | 64 |
| 16.1 Architectural Patterns | 65 |
| 16.1.1 Layered Architecture | 65 |
| 16.1.2 Repository Architecture | 67 |
| 16.1.3 Client-server Architecture | 68 |
| 16.1.4 Pipe and filter architecture (Event-Driven) | 68 |
| 16.1.5 MVC Pattern | 69 |

1 Data Models and its Implementations

1.1 Conceptual Data Model

The conceptual data model describes the database at a very high level and is useful to understand the needs or requirements of the database. It is this model, that is used in the requirement-gathering process i.e. before the Database Designers start making a particular database. One such popular model is the entity/relationship model (ER model). The E/R model specializes in entities, relationships, and even attributes that are used by database designers. In terms of this concept, a discussion can be made even with non-computer science(non-technical) users and stakeholders, and their requirements can be understood.

Entity-Relationship Model(ER Model): It is a high-level data model which is used to define the data and the relationships between them. It is basically a conceptual design of any database which is easy to design the view of data.

Components of the ER model:

- **Entity:** An entity is referred to as a real-world object. It can be a name, place, object, class, etc. These are represented by a rectangle in an ER Diagram.
- **Attributes:** An attribute can be defined as the description of the entity. These are represented by Ellipse in an ER Diagram. It can be Age, Roll Number, or Marks for a Student.
- **Relationship:** Relationships are used to define relations among different entities. Diamonds and Rhombus are used to show Relationships.

1.1.1 Characteristics

- Offers Organization-wide coverage of the business concepts.
- This type of Data Models are designed and developed for a business audience.
- The conceptual model is developed independently of hardware specifications like data storage capacity, location or software specifications like DBMS vendor and technology. The focus is to represent data as a user will see it in the “real world.”

1.2 Representational Data Model

This type of data model is used to represent only the logical part of the database and does not represent the physical structure of the database. The representational data model allows us to focus primarily, on the design part of the database. A popular representational model is a Relational model. The relational Model consists of Relational Algebra and Relational Calculus. In the Relational Model, we basically use tables to represent our data and the relationships between them. It is a theoretical concept whose practical implementation is done in Physical Data Model.

The advantage of using a Representational data model is to provide a foundation to form the base for the Physical model.

1.2.1 Characteristics

- Represents the logical structure of the database.
- Relational models like Relational Algebra and Relational Calculus are commonly used.
- Uses tables to represent data and relationships.
- Provides a foundation for building the physical data model.

1.3 Physical Data Model

The physical Data Model is used to practically implement Relational Data Model. Ultimately, all data in a database is stored physically on a secondary storage device such as discs and tapes. This is stored in the form of files, records, and certain other data structures. It has all the information on the format in which the files are present and the structure of the databases, the presence of external data structures, and their relation to each other. Here, we basically save tables in memory so they can be accessed efficiently. In order to come up with a good physical model, we have to work on the relational model in a better way. Structured Query Language (SQL) is used to practically implement Relational Algebra.

This Data Model describes HOW the system will be implemented using a specific DBMS system. This model is typically created by DBA and developers. The purpose is actual implementation of the database.

1.3.1 Characteristics

- The physical data model describes data need for a single project or application though it maybe integrated with other physical data models based on project scope.
- Data Model contains relationships between tables that which addresses cardinality and nullability of the relationships.
- Developed for a specific version of a DBMS, location, data storage or technology to be used in the project.
- Columns should have exact datatypes, lengths assigned and default values.
- Primary and Foreign keys, views, indexes, access profiles, and authorizations, etc. are defined

2 Relational Model

- **Tulajdonságtípus:**

- Azonos szerepű tulajdonságok **absztrakciója**

- **Egyedtípus:**

- Azonos tulajdonságtípusokkal rendelkező egyedek **absztrakciója**

- **Kapcsolattípus:**

- Két vagy több egyedtípus között fennálló, jól meghatározott viszony

Tulajdonságtípus lehet:

- **Összetettség szerint:**

- atomi
- összetett

- **egyszerre felvett értékek szerint:**

- egyértékű
- halmazértékű

- **megjelenés szerint:**

- tárolt
- származtatott

Kapcsolat:

- **Szorossága:**

- kötelező
- félíg kötelező (egy oldalnak)
- opcionális

- **Fokszám:**

- hány egyedtípus vesz részt

Conceptual, logical and physical modelling levels separate.

D domain is a set of atomic values (including NULL)

- name
- type
- format
- constraints on values

The relational schema R : $R(A_1, A_2, \dots, A_n)$

- R is the name of the schema
- A_1, A_2, \dots, A_n are the attributes

The A_i attribute can have values from the D_i domain (the domain contains the possible values of the attribute)

The degree of a relation is the number of attributes in the realtion.

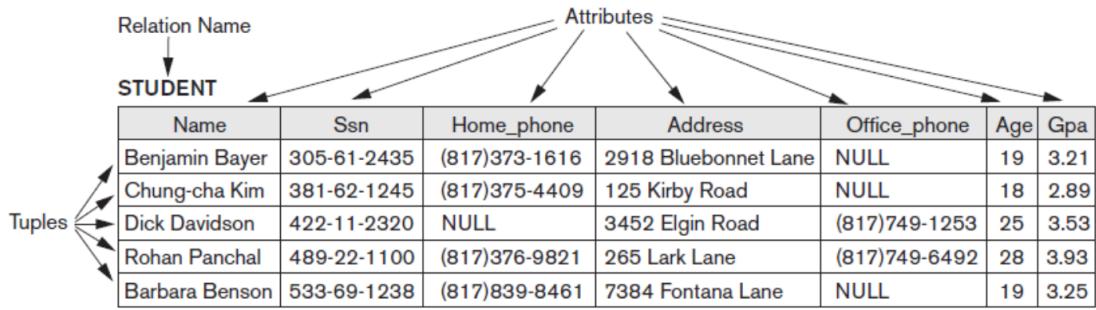


Figure 3.1
The attributes and tuples of a relation STUDENT.

Figure 1: Relational model overview

2.1 Definition of the Relation

R 's defining domains Descartes-product's subset

Az R -et meghatározó tartományok Descartes szorzatának a részhalmaza

$$r(R) \subset (dom(A_1) \times dom(A_2) \times \dots \times dom(A_n))$$

2.1.1 The Other Definition

- **A reláció**

- Az $R(A_1, A_2, \dots, A_n)$ relációséma egy r relációja – amit szokás $r(R)$ -rel is jelölni – elem n -eseknek egy halmaza: $r = \{t_1, t_2, \dots, t_m\}$
- minden t_i elem ($1 \leq i \leq m$) n darab értéknek egy rendezett listája: $t_i = \langle v_{i1}, v_{i2}, \dots, v_{in} \rangle$, ahol
- minden v_{ij} érték ($1 \leq j \leq n$) vagy A_j tartományának az eleme.

The order of the tuples in a relation cannot be interpreted since they are part of a set.

A record's values must be atomic, so they cannot be complex or set values, therefore every relation is in 1NF.

2.2 The NULL value

- We do not know if the value exists
- The value exists but we do not know it
- The value does not exist for the given incidence

3 Integrity Constraints

3.1 Domain Constraint

The domain constrain states that for every record for every value belonging to the A attribute, it must originate from $\text{dom}(A)$, and these values must be atomic

Data types:

- numerical
- integer
- real
- character
- logic
- string
- date
- other special data types (time, currency, timestamp)

3.2 Key Constraints

By definition in a relation every record is different, therefore in a relation there is no 2 records where all the attribute values are the same.

3.2.1 Superkey

Az R relációsémának létezik egy olyan SK attribútumhalmaza, amely olyan tulajdonságú, hogy tekintve R bármelyik r relációját, az adott relációban nincs két olyan rekord, amelynek az értékei azonosak lennének ezen SK attribútumokra vonatkozóan. Azaz bármely két különböző $t1$ és $t2$ rekordot kiválasztva R egy r relációjából: $t1[SK] \neq t2[SK]$. minden ilyen SK attribútumhalmaz az R relációséma szuperkulcsa.

Every relation has a superkey: the set of all the attributes.

3.2.2 Key

Egy R relációséma K kulcsa R -nek egy olyan szuperkulcsa, amelyből egy A attribútumot elhagyva, az így kapott K' attribútumhalmaz már nem szuperkulcsa R -nek.

Satisfies the following two condition:

- Bármilyen relációt tekintve, a reláció két különböző rekordjának nem lehetnek azonosak a kulcsban szereplő attribútumokhoz tartozó értékei.
- Minimális szuperkulcs, azaz egy olyan szuperkulcs, amelyből nem tudunk úgy eltávolítni egyetlen attribútumot sem, hogy az egyediségre vonatkozó feltétel továbbra is fennálljon.

K key is simple if it only consists of one attribute, else it is complex.

A relational scheme can have multiple keys, these are candidate keys.

The modeller's job to choose a **primary key** from the candidate keys, its values will identify each row in the relation

We can put a *unique* constraint to the other candidate keys.

3.3 Entity Integrity Constraint (Egyedintegritasi megszoritas)

It states that no single primary key value can be NULL value. If the primary key is complex, none of its components can be NULL value.

3.4 Referential Integrity Constraint (Hivatkozasintegritasi megszoritas)

A hivatkozási integritási megszorítást két reláció között értelmezzük, és a két relációban lévő rekordok között konzisztencia megteremtése érdekében használjuk.

Egy R_1 relációséma FK -val jelölt attribútumhalmaza **külső** (idegen) **kulcsa** R_1 -nek, amely hivatkozik az R_2 relációsémára, ha eleget tesz a következő feltételeknek:

- Az FK -beli attribútumoknak és az R_2 PK -val jelölt elsődleges kulcsattribútumainak páronként azonos a tartománya; ekkor azt mondjuk, hogy az FK attribútumok hivatkoznak az R_2 relációsémára.
- Bármely $r_1(R_1)$ aktuális állapotának egy t_1 rekordjában egy FK -beli érték vagy megjelenik egy $r_2(R_2)$ aktuális állapotának valamely t_2 rekordjában PK értékeként, vagy az értéke NULL. Az előbbi esetben $t_1[FK] = t_2[PK]$, ekkor azt mondjuk, hogy a t_1 rekord hivatkozik a t_2 rekordra.

Ha e két feltétel teljesül, egy hivatkozási integritási megszorítás áll fenn R_1 -ról R_2 -re vonatkozóan.

Egy relációs adatbázisséma minden integritási megszorítását meg kell határozni.

3.5 Other Type of Constraints

- NULL constraint: Can an attribute's values be NULL
- Semantic integrity constraints: Enforced by software, trigger
- Dependency between data: *functional dependency*, *multivalued dependency*
- State constraints
- Dynamic constraints: E. g. a workers' pay can only grow

4 Relational Database and Relational Database Schema

- **Relational Database Scheme:** S relational database scheme is the $S = \{R_1, R_2, \dots, R_m\}$ relational scheme set and the integrity constraint set.
- **Relational Database (State):** S 's one relational database (state) is such $DB = \{r_1, r_2, \dots, r_m\}$ relation states set, that every r_i is a relation of R_i and every r_i satisfies the integrity constraints given in IC

4.1 Operations on a Relation

- Query
- Insert
- Modify
- Delete

Listing 1: Combining queries on employees and departments

```
1 SELECT * FROM employees e, departments d
2 WHERE e.email = 'SSTILES'
3   AND e.department_id = d.department_id
4 UNION ALL
5 SELECT * FROM employees e, departments d
6 WHERE d.department_name = 'Treasury'
7   AND e.department_id = d.department_id;
```

4.2 ACID Properties

- **Atomicity:** All or nothing execution.
- **Consistency:** Data remains valid after transactions.
- **Isolation:** Concurrent transactions do not interfere with each other.
- **Durability:** Committed data is saved permanently.

5 Object Relational Model

5.1 Weaknesses of RDBMS

- A való világbeli egyedeket szegényesen ábrázolja: A relációk nem tükrözik a való világot
 - Szemantikus túlterhelés
 - Csak reláció van
 - a kapcsolatoknak nem lehet jelentést adni
 - a kapcsolatokat nehéz megkülönböztetni az egyedektől
- Az integritás és a vállalatszintű megszorítások szegényes támogatása
 - Az integritást megszorításokban fejezi ki
 - Sok rendszer nem biztosítja ezek kiképítését, ami miatt be kell építeni őket az alkalmazásba
- Homogén adat struktúra
 - minden sornak ugyanazok az attribútumai
 - Az oszlop minden értékének ugyanabból a tartományból kell származnia

- A mezők értékei csak atomiak lehetnek
- Korlátozott műveletek
 - Ami az SQL specifikációban van
 - Pl: GIS-ben pontok, vonalak, poligonok vannak tárolva és a művelet szükséges a távolság, a metszet vagy a tartalmazás lekérdezésére
- A rekurzív lekérdezéseket nehezen kezeli
- Nehézkessék a séma-változtatások

5.2 ORDBMS

- A relációs adatbázisok beépítették az OO világot
 - Típus rendszer
 - Egységezés
 - Öröklés
 - Polimorfizmus
 - A metódusok dinamikus kötése
 - Összetett objektumok (nem 1NF-ben lévő objektumok)
 - Objektum azonosító.

Advantages:

- Az újrafelhasználás és a megosztás; Pl: A GIS-ben a távolságot egyszer definiáltuk, akkor bárki újra hívhatja
- Megtartja a relációs tudást

Disadvantages:

- Összetett és drága

5.3 User Defined Type (UDT)

Normális eseten egy tábla oszlopának az adattípusa, egy SQL-ben meghívott rutin egy SQL változójának a típusa vagy egy külsőleg meghívott SQL rutin egy paramétere.

A felhasználó által definiált típus egy olyan típus, amely nincs beépítve az adatbázisrendszerbe vagy a programozási nyelvbe, de egy alkalmazásfejlesztés részeként lehet definiálni, amely gyakran a viselkedésének a leírásával is jár.

5.3.1 Distinct Type

Egy egyszerű beépített adattípuson alapszik, mint pl. integer, de ezeknek az értékeit nem lehet közvetlenül összekeverni az eredeti alaptípussal (cast szükséges).

Első osztályú típusok, azaz lehet őket használni oszlopdefinícióban, változó deklarációban, stb, mint bármely más SQL beépített típust.

```
1 CREATE TYPE shoe_size AS INTEGER FINAL;
```

A *Final* kulcsszót meg kell adni a distinct type definíciókban. Adjuk meg a FINAL kulcsszót, ha a típusnak nem lehet altípusa.

5.4 Structured UDT

Belső struktúrája van

```
1 Create type address
2 (Street_name varchar2(50),
3 Apartment_number varchar2(5),
4 City varchar2(50),
5 Country varchar2(50),
6 Postal_code varchar2(10));
```

Két fő jellemző:

- Adatok, amelyeket alapvetően tárolnak
- És műveletek, amelyeket az adatokon lehet végrehajtani. A kódjuk implementációját a típus definiálója adja meg.

Attribútumok

- minden attribútumnak van egy egyszerű típusa (amely nincs korlátozva az SQL beépített atomi típusaira)
- Egy típus attribútumainak kollekcióját a típus reprezentációjának hívják (representation rész)
- A Java objektumok mezőinek felel meg az SQL strukturált típusok attribútumai.

5.4.1 Viselkedés és szemantika

Rutinok segítségével lehet megadni őket (metódusok, eljárások, függvények)

Az SQL lehetővé teszi, hogy a típusok tervezői viselkedést adjanak meg valamilyen nyelven megírt rutinok segítségével

5.4.2 Öröklődés

Az altípus egy olyan adattípust ír le, amely birtokol minden olyan jellemzőt, amellyel egy másik típus

A szupertípus olyan típust ír le, amelynek a jellemzőit egy altípus birtokolja.

5.4.3 Observer és Mutator metódusok

Egy strukturált típus minden attribútumának van két beépített, a rendszer által definiált metódusa:

- **Observer** metódus: Az attribútum értékével tér vissza.
- **Mutator** metódus: lehetővé teszi, hogy az attribútum értéke változzon

5.4.4 Konstruktorok

A típus egy példányát hozza létre, más néven inicializáló metódus.

6 XML Databases

Extensible Markup Language (XML) is a standard for storing and transporting data. Unlike relational databases which store data in rigid rows and columns, XML databases are designed to handle **semi-structured data**.

6.1 Key Characteristics

- **Hierarchical Structure:** Data is stored in a tree-like structure (DOM - Document Object Model) consisting of a root element, child elements, and attributes.
- **Self-Describing:** The schema (tags) travels with the data. A document can be understood without referencing an external catalog (though XSD schemas are used for validation).
- **Flexibility:** New fields (tags) can be added without restructuring the entire database.

6.2 Data-Centric vs. Document-Centric

Understanding the nature of the data is the first step in XML database design.

1. Data-Centric XML:

- Used for data transport (e.g., SOAP messages, REST API responses).
- Highly structured with predictable fields.
- The order of sibling elements often does not matter.
- *Example:* Flight schedules, Stock quotes.

2. Document-Centric XML:

- Used for mixed content (text with markup).
- The order of elements (structure) is critical to the meaning.
- *Example:* A book, a legal contract, or a medical record (HL7).

6.3 Database Architectures

There are two primary ways to implement an XML database system.

6.3.1 1. XML-Enabled Databases (XEDB)

These are traditional Relational Databases (like Oracle, SQL Server, PostgreSQL) that have been extended to handle XML.

- **Storage:** They often store XML in a generic CLOB (Character Large Object) column or a specialized binary XML type.

- **Mapping:** Middleware maps the XML nodes to relational tables ("Shredding").
- **Pros:** Leverages existing ACID properties, security, and transaction management of established RDBMS.
- **Cons:** Performance overhead when parsing deep XML trees; impedance mismatch between the tree model and table model.

6.3.2 2. Native XML Databases (NXD)

Databases specifically designed to store XML documents as the fundamental unit of logical storage (e.g., BaseX, eXist-db, MarkLogic).

- **Storage:** Data is stored physically in a format that closely resembles the logical tree structure (e.g., using persistent DOM or proprietary binary tree formats).
- **No Mapping:** There is no conversion to tables. The database model is the XML document itself.
- **Pros:** Faster retrieval for complex hierarchies; precise document order preservation.
- **Cons:** Lack of standardized tooling compared to SQL; smaller market share.

6.4 Query Languages

Just as SQL is the standard for RDBMS, XML databases utilize specific W3C standards for data retrieval.

6.4.1 XPath (XML Path Language)

XPath is used to navigate through elements and attributes in an XML document. It uses a path-like syntax similar to file systems.

Example Data (books.xml):

```
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

XPath Expressions:

- `/bookstore/book[1]/title`: Selects the title of the first book.
- `//author`: Selects all author nodes anywhere in the document.
- `/bookstore/book[price>35]`: Selects books where the price is greater than 35.

6.4.2 XQuery

XQuery is the functional query language for XML. It is a superset of XPath and allows for complex logic, joining, and reshaping of data (similar to how SQL allows `SELECT ... WHERE`). It relies on the **FLWOR** expression:

- **For:** Iterates over a sequence of nodes.
- **Let:** Binds variables.
- **Where:** Filters results.
- **Order by:** Sorts results.
- **Return:** Constructs the result structure.

XQuery Example:

```
(: Select titles of books costing more than $29 :)
for $x in doc("books.xml")/bookstore/book
where $x/price > 29
order by $x/title
return <result>{ $x/title }</result>
```

6.5 Summary comparison

| Feature | Relational DB | Native XML DB |
|-----------------------|------------------------|--------------------------------|
| Data Model | Tables, Rows, Columns | Trees, Nodes, Elements |
| Structure | Rigid Schema | Semi-structured / Schema-less |
| Order | Unordered sets | Intrinsic order matters |
| Query Language | SQL | XQuery / XPath |
| Integrity | Referential (PK/FK) | Integrity ID/IDREF (weaker) |

Table 1: Comparison of Relational and Native XML Databases

7 NoSQL Databases

Két fő ok miatt születtek meg:

- A nagy adattömeg kezelése kikényszerítette azt, hogy klaszterezéssel kapcsoljanak össze gépeket, és így nagy hardver platformokat építsenek.
- Ez az igény azt is megnehezítette, hogy az alkalmazáskódok jól működjenek a relációs modellel.

A két ok miatt használunk NoSQL adatbázist:

- Az alkalmazásfejlesztés termelékenysége miatt: sok idő és erőfeszítés megy el arra, hogy a memóriastruktúrákat leképezzék relációs adatbázisra. A NoSQL adatbázisok olyan adatmodellt biztosítanak, amelyek jobban alkalmazkodnak az alkalmazások

szükségleteihez. Leegyszerűsödik az adatbázis elérése, kevesebb kódot kell írni, nyomkövetni és javítani.

- A nagymennyiségek miatt: ma megéri több adatot tárolni és sokkal gyorsabban feldolgozni. Relációs adatbázissal ez drága vagy lehetetlen. A relációs adatbázisokat úgy terveztek, hogy egy gépen fussenak, azonban ma már általában gazdaságosabb sok kisebb, olcsóbb gép klaszterén végezni a nagymennyiségek adat feldolgozását. Sok NoSQL adatbázist úgy terveztek, hogy klasztereken futnak.

A fő kategóriák:

- Kulcs-érték (key-value) adatbázisok
- Dokumentum adatbázisok
- Oszlopcsalád (column-family) adatbázisok
- Gráf adatbázisok
- Objektum adatbázisok
- XML adatbázisok

Vannak nem tiszta adatbázisok is, amelyek kevernek két kategóriát.

7.1 Aggregátum adatmodellek

A kulcs-érték, a dokumentum és az oszlopcsalád adatmodellek együtt aggregátum adatmodelleknek hívjuk, a közös jellemzőik miatt.

Az aggregátumban kulcs-érték párok vannak, ahol az érték lehet dokumentum, egyszerű adat, vagy valamilyen összetettebb struktúra, mint halmaz, lista, stb.

Azt támogatja, hogy gyakran akarunk olyan adatokon dolgozni, amelyeket olyan unióba akarunk szervezni, amelynek bonyolultabb a struktúrája, mint amelyet a listák és a rekordstruktúrák egymásba ágyazása lehetővé tenne.

Az **aggregátum** kapcsolódó objektumok egy olyan gyűjteménye, amelyeket egy egységeként szeretnénk kezelni. Ez egyben az adatmódosítás és a konzisztencia menedzsment egysége. Általában atomi műveletekkel szeretjük módosítani az aggregátumokat és aggregátumban kifejezve szeretünk kommunikálni az adattárolóval.

Az aggregátumok megkönnyítik az adatbázisoknak a klaszteren való műveletek kezelését, mert az aggregátum a replikáció és a sharding természetes egysége.

Az aggregátumok megkönnyítik a programozók dolgát is, mert az gyakran az aggregátumokon keresztül módosítják.

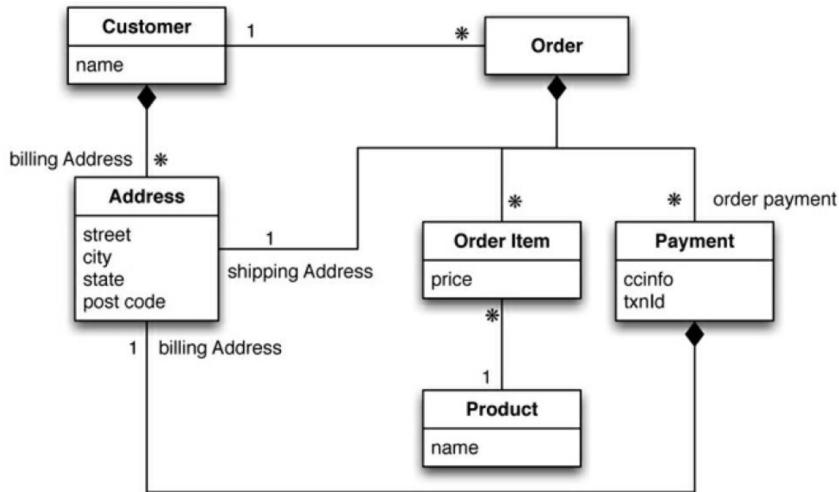


Figure 2: Example of an aggregate structure in a NoSQL context.

Listing 2: Aggregate stored as nested JSON

```

1  {
2      "customer": {
3          "id": 1,
4          "name": "Martin",
5          "billingAddress": [
6              {
7                  "city": "Chicago"
8              }
9          ],
10         "orders": [
11             {
12                 "id": 99,
13                 "customerId": 1,
14                 "orderItems": [
15                     {
16                         "productId": 27,
17                         "price": 32.45,
18                         "productName": "NoSQL Distilled"
19                     }
20                 ],
21                 "shippingAddress": [
22                     {
23                         "city": "Chicago"
24                     }
25                 ],
26                 "orderPayment": [
27                     {
28                         "ccinfo": "1000-1000-1000-1000",
29                         "txnid": "abelif879rft",
30                         "billingAddress": {
  
```

```

31         "city": "Chicago"
32     }
33   }
34 ]
35 }
36 ]
37 }
38 }
```

Nincs univerzális válasz arra a kérdésre, hogy hol húzzuk meg az aggregátumok határát. Csak attól függ, hogy hogyan fogjuk módosítani az adatot.

Ha a customer-t az orders-sel együtt fogjuk módosítani, akkor egy aggregátummal fogunk dolgozni.

Ha módosításkor egy megrendelésre fókuszálunk, akkor érdemes szétválasztani az aggregátumokat.

Gyakran előfordul, hogy egy alkalmazás minden két megközelítést is használná.

7.1.1 Előnyei

Nagyszerűen segít a klaszteren való futtatást, ahol minimalizálnunk kell az adatok lekérdezéshez szükséges csomópontok számát. Az aggregátumokkal egyértelműen megmondjuk az adatbázisnak, hogy mely adatokat fogjuk együtt módosítani. Ezeknek az adatoknak egy csomópontron kell lenniük.

7.1.2 Hátrányai

- Gyakran nehéz meghúzni a határt az aggregátumok között, különösen akkor, amikor ugyanazt az adatot több különböző esetben használjuk.
- Az aggregátum struktúra sok típusú adatlekérdezésben módosításban segíthet, azonban más típusú adatlekérdezésben módosításban gátolhat.

Gyakran azt mondják, hogy a NoSQL adatbázisok nem támogatják az ACID tranzakciókat, és így felalldozza a konzisztenciát.

Általában igaz, hogy az aggregátumorientált adatbázisoknak nincs olyan ACID tranzakciójuk, amely több aggregátumra kiterjed.

Helyette **egy aggregátum egyetlen atomi módosítását** támogatják. Ez azt jelenti, hogy ha több aggregátumot szeretnénk atomi módon módosítani, azt nekünk az alkalmazásunkban kell kezelni.

7.2 Kulcs- érték és dokumentum adatmodellek

Mindkét típusú adatbázis sok aggregátumból épül fel, ahol minden aggregátumnak van egy kulcsa vagy egy azonosítója, amelyet az adat elérésére használunk.

Kulcs-érték adatbázisok

- Az aggregátum **átlátszatlan** az adatbázisban. Az átlátszatlanság előnye az, hogy azt tárolhatunk az aggregátumban amit csak szeretnénk. Az adatbázisnak lehet, hogy van valamilyen méretkorlátozása, de ez több, mint ami a korlátozná a szabadságunkat.
- Egy aggregátumot csak a kulcsán keresztül kereshetünk ki.

Dokumentum adatbázisok

- Képes látni az aggregátum **struktúráját**. A dokumentum adatbázis korlátosokat adhat arra nézve, hogy milyen struktúrákat és típusokat helyezhetünk el egy aggregátumba. Cserébe az elérésnél nagyobb rugalmasságot biztosít.
- Lekérdezéseket írhatunk az aggregátum mezői alapján, kinyerhetjük az aggregátumok részeit a teljes aggregátum helyett, és indexeket hozhatunk létre az aggregátum tartalma alapján.

7.3 Oszlopcsalád tárak

A legtöbb (relációs) adatbázisnak a tárolási alapegysége a sor. Ez az írási teljesítményt támogatja. Azonban sok olyan forgatókönyv van, ahol az írás kevés, de gyakori az olyan lekérdezés, ahol sok sornak néhány oszlopát kérjük le. Az ilyen esetekben a legjobb megoldás az, ha a sorokhoz oszlopok csoportjait tároljuk, és ezek a csoportok lesznek a tárolás egységei. Emiatt hívják őket oszloptáraknak.

A legkönnyebb talán úgy megérteni az oszlopcsalád modellt, mint egy kétszintű aggregátumstruktúrát. Úgy, mint a kulcs-érték tárknál, az első kulcsot gyakran a sor azonosítójaként értelmezik, amellyel a keresett aggregátumot meg lehet fogni. A különbség az, hogy az oszlopcsalád tárknál a soraggregátum önmagában részletesebb értékek egy leképezését formázza. Ezeket a második szintű értékeket hívjuk oszlopoknak.

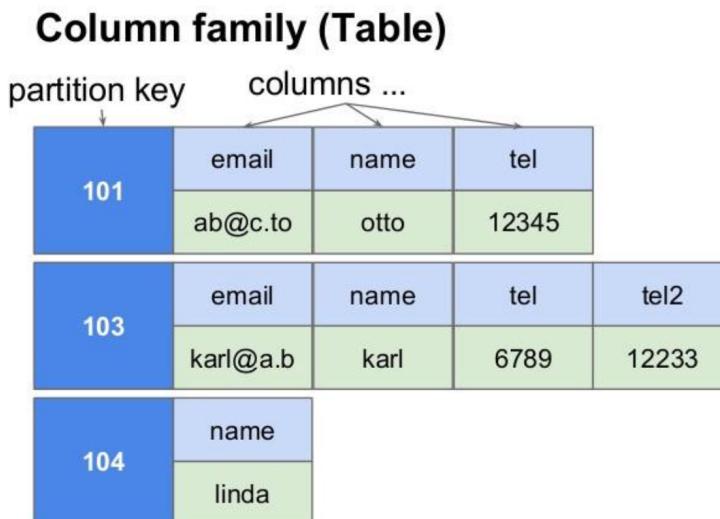


Figure 3: Column-family layout illustrating rows grouped into column families.

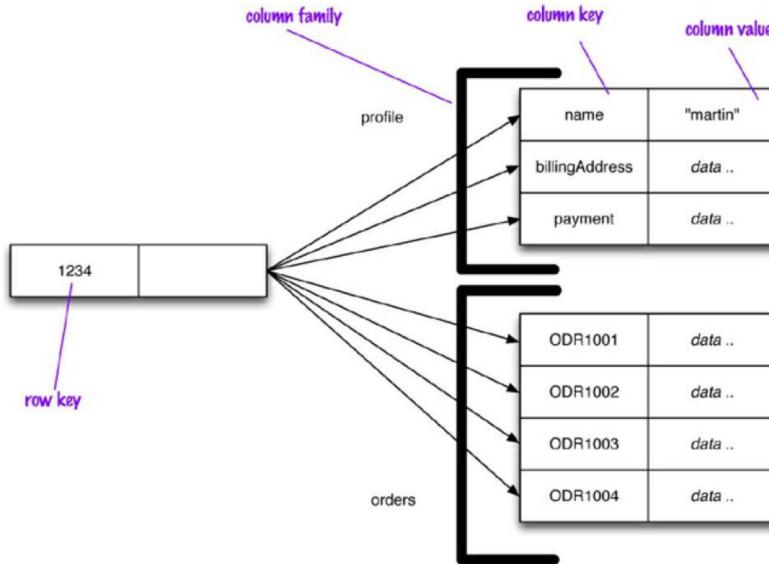


Figure 4: Wide-column data model showing multiple column families per entity.

Az oszlopcsalád adatbázisok az oszlopaikat oszlopcsaládokba szervezik. minden oszlopnak egy oszlopcsalád részének kell lennie. Az oszlop az elérés egysége, azonban feltételezzük, hogy egy oszlopcsaládot általában együtt érjük el.

Az adatokat kétféleképp strukturálhatjuk:

- **Sororientáltan:** minden sor egy aggregátum (pl. customer, amelynek az ID-ja 1234), amelynek az oszlopcsaládjai hasznos adatcsonkokat tartalmaz az aggregátumon belül (profile, order history).
- **Oszloporientáltan:** minden oszlopcsalád egy rekordtípust definiál (pl. customer profiles), ahol minden rekordhoz több sor tartozik. Így minden oszlopcsaládban úgy gondolhatsz egy sorra, mint rekordok összekapcsolása.

Bármilyen oszlophoz bármilyen sort hozzá lehet adni, így a soroknak különböző oszlopkulcsai lehetnek. Új oszlopokat adhatunk a sorokhoz a szokásos adatbázis használat alatt. Azonban az új oszlopcsaládok definiálása ritka, és ehhez gyakran le kell állítani az adatbázist.

A **sovány soroknak (Skinny rows)** kevés oszlopuk van, és sok különböző sorban ugyanazokat az oszlopokat használják. Ebben az esetben az oszlopcsalád egy rekordtípust definiál, minden sor egy rekord, és minden oszlop egy mező.

A **széles sornak (wide row)** sok oszlopa van (akár több ezer), és a soroknak nagyon különböző oszlopai vannak. Ebben az esetben a széles oszlopcsalád egy listát modellez, ahol minden oszlop egy elem a listában.

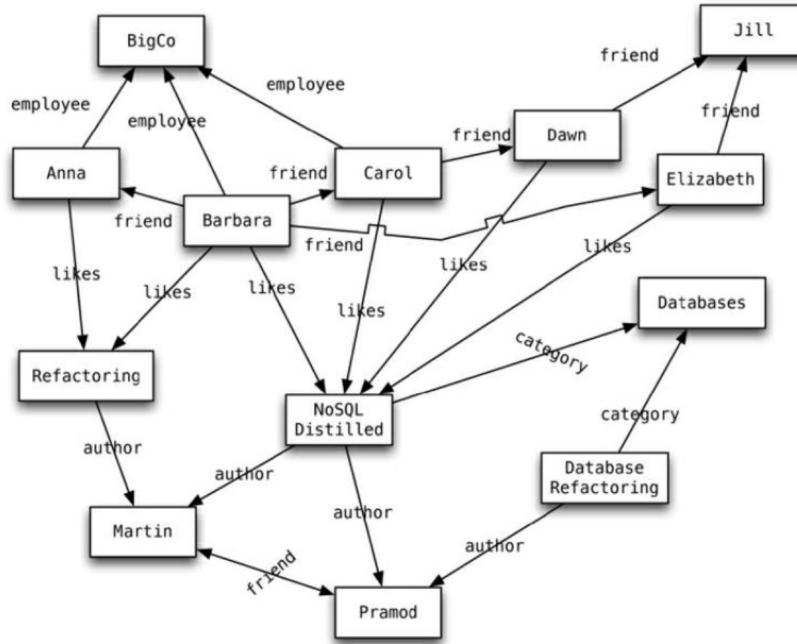
7.4 Gráf adatbázisok

A gráf adatbázisokat az olyan kis rekordok motiválták, amelyek között bonyolult kapcsolatok állnak fenn.

Ebben az értelemben a gráf egy gráfadatstruktúra, élekkel (edge) összekapcsolt csomópon-
tok (node).

Ebben a struktúrában olyan kérdéseket tehetünk fel, mint „keressük azt a könyvet az Adatbázis kategóriában, amelyet olyan valaki írt, akit valamelyik barátom kedvel”.

Ideális megoldás olyan esetben, amikor összetett kapcsolatokat tartalmazó adatokat szeretnénk tárolni, mint a szociális hálók (social networks), vagy terméka jánlások.



A relációs adatbázisok a kapcsolatot külső kulcs segítségével valósítják meg. Az összekapcsolások, amelyek a navigációhoz szükségesek meglehetősen drágák lehetnek, amely azt jelenti, hogy a teljesítmény gyakran gyenge a sok kapcsolattal rendelkező adatmodellekben.

A gráfadatbázisok navigálása a kapcsolatokon keresztül nagyon olcsó. A gráfadatbázisok a navigálás munkájának a nagyrészét beszűráskor végzi el és nem a lekérdezéskor. Ez természetesen akkor kifizetődő, amikor a lekérdezés teljesítménye sokkal fontosabb, mint a beszűrésé.

Ezek az adatbázisok főleg egy gépen futnak és nem klasztereiken elosztva.

7.5 Sémamentes adatbázisok

Egyszerűen azt tárolhatjuk, amire szükségünk van. Lehetővé teszik, hogy egyszerűen módosítsuk az adattárunkat a projektünk előrehaladtával. Ha új dolgot fedezünk fel, egyszerűen hozzáadhatjuk. Ha rájövünk, hogy valamit nem szükséges tárolnunk, akkor egyszerűen törölhetjük.

Egyszerűvé teszik, hogy nemegységes adatokkal (olyan adatokkal, amelyeknél a rekordok más mezőből épülnek fel) dolgozzuk. Lehetővé teszik, hogy a rekordok pontosan azt tartalmazzák, amire szükségünk van, nem többet és nem kevesebbet.

A probléma: tény, hogy amikor adatokat használó programot írunk, akkor a program majdnem minden implicit séma valamilyen formájában.

Attól, hogy az adatbázisunk sémamentes, általában van egy implicit séma. Ez az implicit séma olyan feltételezések halmaza, amelyek az adatokat manipuláló kódban lévő adatstruktúráról szólnak.

Az alkalmazáskódban szereplő implicit séma eredményez néhány problémát. Ha meg akarjuk érteni, hogy mit jelent az adat, akkor az alkalmazáskódunkban kell ásni.

7.6 Elosztási modellek

A NoSQL adatbázisok elsődleges érdekeltsége az a képesség volt, hogy az adatbázis egy nagy klaszteren fusson.

Az aggregátum megközelítés jól illeszkedik a skálázáshoz, mert az aggregátum az elosztáshoz használt természetes egység.

Két út van az adatok elosztásához:

- **Replikáció:** A replikáció ugyanazt az adatot több csomópontra helyezi.
- **Sharding:** A sharding különböző adatokat helyez a csomópontokra.

A replikáció és a sharding ortogonális technikák: lehet használni az egyiket vagy akár mindkettőt.

A replikáció két formában jelenhet meg: **master-slave** és **peer-to-peer**.

Egyedülálló szerver:

- A legegyszerűbb elosztási modell
- nincs elosztás
- szeretjük, mert kiküszöböli az összes komplexitást, amellyel a többi modell megvalósítás jár
- A gráf adatbázisok tartoznak ebbe a kategóriába, mivel legjobban az egy szerveres konfigurációban működnek.
- Ha az adathasználatunk leginkább aggregátumok feldolgozásáról szól, akkor egy egyszerveres dokumentum vagy kulcs-érték tár jó választás lehet, mert egyszerűbbek a fejlesztőknek.

7.6.1 Sharding

A sharding különböző adatokat különböző csomópontokra tesznek, minden csomópont saját maga végzi az írást és az olvasást.

Az aggregátumokat úgy tervezzük, hogy olyan adatokat tartalmazzanak, amelyeket általában együtt érünk el, így az aggregátum lesz az elosztás nyilvánvaló egysége.

Néhány tényező javíthatja a teljesítményt:

- Az adatokat ahhoz közel helyezzük el, ahol használni fogják.

- Próbáljuk meg úgy elhelyezni az aggregátumokat, hogy egyenlően legyenek elosztva a csomópontok, amelyek mindenike egyenlő terhelést kap.
- Néhány esetben hasznos, ha az aggregátumokat együtt helyezzük el, ha úgy gondoljuk, hogy sorban fogjuk felolvasni őket (A Bigtable azt javasolja, hogy a sorokat abc sorrendben tároljuk, és a webcímeket a megfordított domain neveik alapján rendezzük (pl: com.martinfowler). Így több oldal adatát érhetjük el egyszerre, amivel javítjuk a feldolgozás hatékonyságát.)
- Sok NoSQL adatbázis ajánl **auto-sharding**-ot, ahol az adatbázis vállalja a felelősséget annak, hogy kiutalja a shard-ot az adathoz és biztosítsa, hogy az adat a megfelelő shard-ra kerül.

A sharding különösen értékes a teljesítmény tekintetében, mert az írási és az olvasási teljesítményt is növelheti. A sharding horizontálisan skálázható írást biztosít.

7.6.2 Master-Slave replikáció

- Több csomópontra másoljuk az adatot.
- Egy csomópontot kijelölünk master vagy elsődleges csomópontnak. A master a hiteles forrása az adatoknak és általában ő a felelős az adatok módosításáért. A többi csomópont slave vagy másodlagos csomópont. A replikációs folyamat szinkronizálja a slave-eket a master-rel.
- Master-slave replikáció leginkább akkor hasznos, amikor olvasás intenzív adathalmazt skálázunk. Több olvasási kérés kiszolgálásához horizontálisan skálázhatunk, ha több slave csomópontot adunk a rendszerhez és biztosítjuk, hogy minden olvasási kérés a slave-ekhez fussen be.
- Nem jó választás, ha az adathalmazunkon sok írási forgalom történik. Korlátozva vagyunk a master írási képességével és azzal a képességgel, hogy továbbadja a módosításokat.
- A másik előnye az olvasási rugalmasság: ha a master meghibásodik, a slave-k még mindig tudják kezelni az olvasási kéréseket. Ez akkor hasznos, ha a legtöbb adatelérés olvasás. A master meghibásodása lehetetlenné teszi az írások kezelését addig, amíg a master helyre nem áll vagy egy másik mastert ki nem jelölünk. Mivel a slave-k a master másolatai, a master meghibásodása utáni helyreállás felgyorsul, mivel egy slave-t gyorsan ki lehet jelölni új masternek.
- A mastert ki lehet jelölni kézzel és automatikusan.
 - A kézzel való kijelölés tipikusan azt jelenti, hogy amikor konfiguráljuk a klasztert, akkor egy csomópontot masternek konfigurálunk.
 - Az automatikus kijelölésnél létrehozzuk a csomópontok klaszterét és ők választanak ki egyet maguk közül masternek.
- A hátránya az inkonzisztencia.
 - Az a veszély, hogy különböző kliensek különböző slave-keket olvasva, **különböző értékeket** fognak látni, mivel nem minden változás adódik tovább azonnal a slave-knek. Legrosszabb esetben egy kliens nem tud olvasni egy írást, amit most készült.

- Ha master-slave replikációt használunk **forró mentéshez**, ez az eset akkor is fennáll, mivel ha a master meghibásodik, minden olyan módosítás, amelyet nem továbbított a mentésnek, elveszik.

7.6.3 Peer-to-Peer replikáció

- minden csomópont fogadhat írást és olvasást minden adatra.
- minden csomópontnak egyenlő súlya van, mindenki fogadhat írást, és ha valamelyiküket elvesztjük, az nem akadályozza meg az adattár elérését.
- A meghibásodásokat könnyen áthidalhatjuk úgy, hogy az adatokhoz végig hozzáférünk.
- Könnyen adhatunk hozzá új csomópontot, hogy javítsuk a teljesítményt.
- A legnagyobb bonyodalom a konzisztencia. Ha két különböző helyet írhatunk, akkor megkockázatjuk azt, hogy két felhasználó ugyanazt a rekordot ugyanabban az időben próbálja módosítani, amely írás-írás konfliktust okoz. Az olvasási inkonzisztencia is vezethet problémákhöz, de azok legalább relatívan átmenetiek.

7.6.4 A sharding és a replikáció kombinációja

Ha master-slave replikációt és shardingot használunk egyszerre, az azt jelenti, hogy több masterunk van, de egy adatelem csak egy masteren van. Konfigurációtól függően kiválaszthatsz egy csomópontot masternek és a többit slave-nek vagy választhat sz több mastert és több slavet.

Peer-to-peer replikáció és sharding használata az oszlopcsalád adatbázisoknál szokásos stratégiá.

Jó kiindulási pont a peer-to-peer replikációkhoz, ha a replikációs faktor 3, így minden shard 3 csomópontron jelenik meg. Ha egy csomópont meghibásodik, akkor a shard-jai a többi csomópontron felépülnek.

7.6.5 Konzisztencia

Módosítási konzisztencia:

- **Írás-írás konfliktus:** két ember ugyanabban az időpontban ugyanazt az adatelemet módosítja.
- Amikor az írás eléri a szervert, akkor a szerver sorbaállítja (serialize) őket, azaz előönti, hogy melyiket hajtja végre elsőként majd másodikként.
- **Elveszett módosítás:** Martin módosítása után Péter azonnal felülírja az adatelemet, akkor a Martin módosítása elveszett. Konzisztenciahibának látjuk, mert Péter módosítása azon az állapoton hajtódott végre, amire Martin módosított, bár azt várjuk, hogy az eredeti állapoton hajtódjon végre.

Módosítási konzisztencia:

- A konzisztencia kezelését kétféleképp szokták megközelíteni: pessimista kezelés és optimista kezelés.

- A **pesszimista** megközelítésben megelőzzük a konfliktusokat; az **optimista** megközelítésben megengedjük a konfliktusokat, de észrevesszük őket és intézkedéseket teszünk az eltávolításukra.
- Az írási konfliktusokhoz a szokásos **pesszimista** megközelítés az **írási zárak** elhelyezése, azaz ha egy értéket meg akarunk változtatni, előtte zárat kell elhelyezni rajta. A rendszer biztosítja, hogy egyszerre csak kliens tehet zárat egy adatelemre. Ha Martin és Péter zárat kérne, akkor csak Martin (az első) járna sikerrel. Péter így látná Martin írását, mielőtt eldöntené, hogy elvégzi-e a módosítását.
- A szokásos **optimista** megközelítés a **feltételes módosítás**, ahol bármely kliens (amelyik módosítást végez) a közvetlenül a módosítás előtt leteszteli, hogy a módosítandó érték változott-e az utolsó olvasása óta. Ebben az esetben Martin módosítása sikeres lenne, Péteré meghiúsulna. A hibából Péter tudná, hogy újra meg kell néznie az értéket és eldönteni, hogy akar-e módosítani.
- Egy másik **optimista** megközelítés az írás-írás konfliktus kezelésére, amikor **elmentjük minden módosítást** és rögzítjük, hogy konfliktusban állnak. Ez ismerős lehet a verziókezelő rendszerekből. A következő lépésben a két módosítást valahogyan **össze kell olvasztani** (kézzel vagy automatikusan).
- A párhuzamos programozás alapvető velejárója, hogy mérlegelni kell a biztonság (a hibák, mint a módosítási konfliktusok kerülése) és a gyorsaság (gyorsan válaszolunk a kliensnek) között.

Olvasási konzisztencia

- **Logikai konzisztencia:** biztosítja, hogy a különböző adatelemeknek **együtt van értelmük**. A logikai inkonzisztencia elkerülését a relációs adatbázis a tranzakciókkal végzi. Ha Martin két módosítása egy tranzakcióban lenne, a rendszer garantálná, hogy Péter vagy a módosítások előtti vagy a módosítások utáni adatokat olvasná.
- Természetesen nem lehet minden adatot egy aggregátumban elhelyezni, így minden olyan módosítás, amely több aggregátumot érint, hagy egy időablakot, amikor a kliensek inkonzisztens olvasást tudnak végezni. Azt az időtartamot, amikor az inkonzisztencia fennáll **inkonzisztencia ablaknak** hívják. Egy NoSQL rendszernek lehet nagyon rövid inkonzisztenciaablaka: Az Amazon dokumentációja azt írja, hogy a SimpleDB inkonzisztenciaablaka általában kevesebb, mint egy másodperc.
- **Másolási (replikációs) konzisztencia:** biztosítja, hogy ugyanannak az adatelemeleknek ugyanaz az értéke, amikor különböző másolatokról olvassuk.
 - Képzeljük el, hogy van egy utolsó hotelszoba egy adott időpontra. A hotel foglalási rendszere sok csomóponton fut. Martin és Cili együtt (mert ők egy pár) szeretnék lefoglalni a szobát, de Martin Londonban van, Cili Budapesten. Telefonon beszélgetnek. Közben Péter Tokióból lefoglalja a szobát. Ez módosítja a másolt szoba elérhetőségét, de a módosítás Budapestre hamarabb odaér, mint Londonba. Amikor Martin és Cili frissítik a böngészőjüket, hogy lássák, hogy a szoba elérhető-e még, Cili foglaltnak látja, míg Martin szabadnak.
- **Egyszer majd valamikor a jövőben (eventually) konzisztencia:** bármely időben lehetnek a csomópontok másolási inkonzisztensek, de ha nincs több mó-

dosítás, akkor végső soron minden csomópont ugyanarra az értékre fog módosulni.

- **Lejárt (stale):** az adat elavult (amely emlékeztet minket arra, hogy a cache a replikáció egy másik formája)
- **Olvasom az írásom (read-your-writes) konzisztencia,** amely azt jelenti, hogy ha egyszer módosítottunk valamit, akkor azt a módosítást garantáltan látni fogjuk.
 - Előfordulhat a következő szituáció: az írásunkat az egyik csomópont fogadja és a klaszteren néhány perces az inkonzisztenciaablak. Az utolsó írásunk után frissítjük a böngészőket, ami egy másik csomópontra dob át, amelyhez még nem jutott el a módosítás. Így úgy tűnhet, mintha elveszítettük volna az írásunkat.
- Az egyik módja, hogy elérjük az olvasom az írásom (read- your-writes) konziszenciát egy egyébként Egyszer majd valamikor a jövőben (eventually) konzisztens rendszerben, ha biztosítjuk a **munkamenet (session) konzisztenciát:** egy felhasználói munkamenet olvasom az írásom (read-your- writes) konzisztens.
- Több módszer létezik a munkamenetkonzisztencia biztosítására. Az általános és gyakran a legkönnyebb módszer a **ragadós (sticky) munkamenet:** a munkamenet egy csomóponthoz van kötve (ezt hívják session affinity-nek is). A hátránya az, hogy a load balancer nem tudja olyan jól végezni a munkáját.
- A munkamenetkonzisztencia másik megközelítése a verzióbélyeg (version stamps) használata. minden adattábeli érintkezéskor használjuk a munkamenet által látható utolsó időbélyeget. A szerver csomópontoknak biztosítaniuk kell, hogy meglegyenek az a módosítás, amely a megfelelő verzióbélyeget tartalmazza, mielőtt egy kérésre válaszol.

The CAP Theorem

CAP: konzisztencia (consistency), elérhetőség (availability), és partíciótolerancia (partition tolerance)

Az elmélet szerinte csak kettőt lehet megkapni egy rendszerben.

- **Az elérhetőség:** minden egyes olvasás és írás vagy sikeresen feldolgozásra kerül vagy egy hibaüzenetet kap, hogy a műveletet nem lehet végrehajtani.
- A **partíciótolerancia:** Egy rendszer akkor partíció toleráns, ha a kérésre hálózati partíció esetén is helyes választ ad (kivéve a teljes hálózat kiesésének esetét). Ha egy rendszer nem partíció toleráns, a hálózat partíciója esetén semmilyen garanciát nem nyújt a konzisztenciára és a rendelkezésre állásra. A hálózat partíciója annak alapján modellezhető, hogy a hálózat egyik csomópontjából a másikba küldött üzenetekből tetszőleges számú elveszhet. Egy nem összefüggő hálózatban a hálózat egyik komponenséből a másikba küldött minden üzenet elveszik.
- **Konzisztencia:** Egy elosztott rendszer akkor konzisztens, ha bármely időpillanatban egy adategység értékét bármely csomóponttól lekérdezve ugyanazt az értéket kapjuk.

Az egyedülálló szerver nyilvánvaló példa a **CA rendszerre:** a rendszer konzisztens és elérhető, de nem partíciótoleráns. Ebben a világban él a legtöbb relációs adatbázisrendszer.

A CAP theorem valódi lényege: Gyakran azt mondják, hogy az CAP elmélet szerinte “csak kettőt kaphatunk meg a háromból”, a gyakorlatban ez úgy hangzik, hogy ha egy rendszer partíciókból áll, mint az elosztott rendszerek, akkor egyensúlyozni kell a konzisztencia és az elérhetőség között. Ez nem egy bináris döntés, gyakran kis konzisztencia mellett döntünk, hogy jobb elérhetőséget kapjunk. Az eredmény rendszer nem lesz tökéletesen konzisztens és nem lesz tökéletesen elérhető, de egy olyan kombinációja lesz, amely alkalmas a rendszer szükségleteihez.

Gyakran jobb a **konzisztencia** és a **lappangás között egyensúlyozására** gondolni a konzisztencia és az elérhetőség egyensúlyozása helyett. Elosztott környezetben a konzisztenciát úgy javíthatjuk, ha több csomópontot vonunk be az interakcióba (azaz az írásba vagy az olvasásba), azonban minden hozzáadott csomópont növeli az interakció válaszidejét. Az elérhetőségre így úgy gondolhatunk, mint a lappangás olyan korlátozására, amelyet még tolerálni tudunk, azaz ha a lappangás túl magas lesz, feladjuk és az adatot elérhetetlenként kezeljük. Ez illeszkedik a CAP-ban lévő definícióhoz.

BASE

A NoSQL követői azt mondják, hogy a relációs tranzakciók ACID tulajdonságai helyett a NoSQL rendszerek a BASE tulajdonságokat követik (**Basically Available, Soft state, Eventual consistency**) - (**alapvetően elérhető, lágy állapot, egyszer majd valamikor a jövőben konzisztens**)

Quorum-ok

Minél több csomópontot bevonunk egy kérésbe, annál nagyobb az esélyünk, hogy elkerüljük az inkonzisztenciát. Ez természetesen felveti a kérdést: hány csomópontot kell bevonnunk, hogy erős konzisztenciát kapjunk?

Az **írási quorum**-ot a $W > N/2$ egyenlőtlenséggel fejezhetjük ki, ahol W az írásban résztvevő csomópontok száma, N a másolatokba bevont csomópontok száma, amelyet úgy is hívnak, hogy **replikációs faktor**.

Olvasási quorum: hány csomóponttal kell kapcsolatot létesítened ahoz, hogy biztos legyél abban, hogy a legfrissebb adatot kapod. Az olvasási quorum egy kicsit komplikáltabb, mert attól függ, hogy hány csomópont igazolta vissza az írást. Akkor van erősen konzisztens olvasásunk, ha $R + W > N$, ahol R az olvasáskor elérendő csomópontok száma, W csomópont igazolta vissza az írást, és a replikációs faktor N .

Ezt az egyenlőtlenséget a peer-to-peer elosztási modell esetén tartsuk észben. Ha egy master-slave elosztási modellünk van, akkor csak a mastert kell írni, hogy elkerüljük az írás-írás konfliktust és hasonlóan a masterről kell olvasni, hogy elkerüljük az olvasás-írás konfliktust.

8 ER Model

Entity-Relationship (ER) model: Népszerű magasszintű koncepcionális modell
ER diagram: Az ER modellhez kapcsolódó diagramszerű jelölésmód

ER model az adatokat, mint a következőket írja le:

- Egyedek

- Tulajdonságok
- Kapcsolatok

8.1 Egyed

A valós világ tárgya (thing, teremtése, valamije) független létezéssel

Egyed típus - egyed előfordulás

8.2 Attribútumok

Attribútum típus - attribútum előfordulás

Az egyedet leíró tulajdonságok

Attribútumok típusai:

- Összetett - egyszerű (atomic)
- Egyértékű - halmazértékű (többértékű)
- Tárolt - származtatott
- NULL értékek
- Komplex attribútumok (összetett és halmazértékű attribútumok tetszőlege egymásbaágazása)

8.2.1 Egyedtípus

Egyedek olyan halmaza vagy kollekciója amelyeknek ugyanolyan attribútumaik vannak

8.2.2 Azonosító

Attribútumok, amelyeknek az értéke egyedi egy egyedhalmaz egyedelőfordulásaiban

Azonosító attribútum: Az egyediség tulajdonságának az egyedtípus minden egyedhalmzára fenn kell állni. Egy egyszerű (vagy egy összetett) attribútum lehet

8.2.3 Értékhalmaz (vagy értéktartomány)

Az egyedek attribútumaihoz rendelhető értékek tartományát határozza meg.

8.2.4 Kapcsolat

Ha egy attribútum egy másik egyedtípusra hivatkozik, akkor a hivatkozást attribútum helyett kapcsolattal reprezentáljuk

Kapcsolattípus (R) : E_1, E_2, \dots, E_n egyedtípusok közötti viszony

Az egyedtípusok egyedelőfordulásai között létrejövő asszociációkat definiálja

Kapcsolat előfordulás r_i

- minden r_i n darab önálló egyeddel van kapcsolatban (e_1, e_2, \dots, e_n)

- minden ri -beli e_j egyed az E_j egyedhalmaz egy tagja

A kapcsolat foka

- A résztvevő egyedtípusok száma
- Bináris, ternáris

Kapcsolat, mint attribútum

Egy bináris (1:1 vagy 1:n) kapcsolatot kifejezhetünk attribútumként

Szerepkör

A szerepkör kifejezi, hogy a résztvevő egyed milyen szerepet játszik a kapcsolatban

Rekurzív kapcsolat

Egy kapcsolattípusban ugyanaz az egyedtípus többször vesz részt különböző szerepkörökkel

A szerepkörnevet meg kell adni

8.2.5 Bináris kapcsolattípusok strukturális megszorításai

Számosság

Azon kapcsolat előfordulások maximális számát határozza meg, amelyben az egyed részt vehet

1:1, 1:n, n:m

Részvételi megszorítás

Megadja, hogy egy egyed létezése függ-e attól, hogy kapcsolatban áll egy másik, a kapcsolattípuson elérhető egyeddel

A kapcsolatelőfordulások minimális számát határozzák meg

Típusai: **totális** (létezésfüggőség) és **részleges**

8.3 Gyenge egyedtípusok

Nincs saját azonosító attribútumuk: Azonosítása egy másik egyedtípus egyedének meghatározásával van kapcsolatban.

Azonosító kapcsolat: A gyenge egyedtípust a tulajdonosával összekötő kapcsolat

A gyenge egyedtípus minden résztvevője a kapcsolatnak

Részleges kulcs (diszkriminátor): egyértelműen azonosítják azokat a gyenge egyedeket, amelyek ugyanazon tulajdonos egyedhez kapcsolódnak

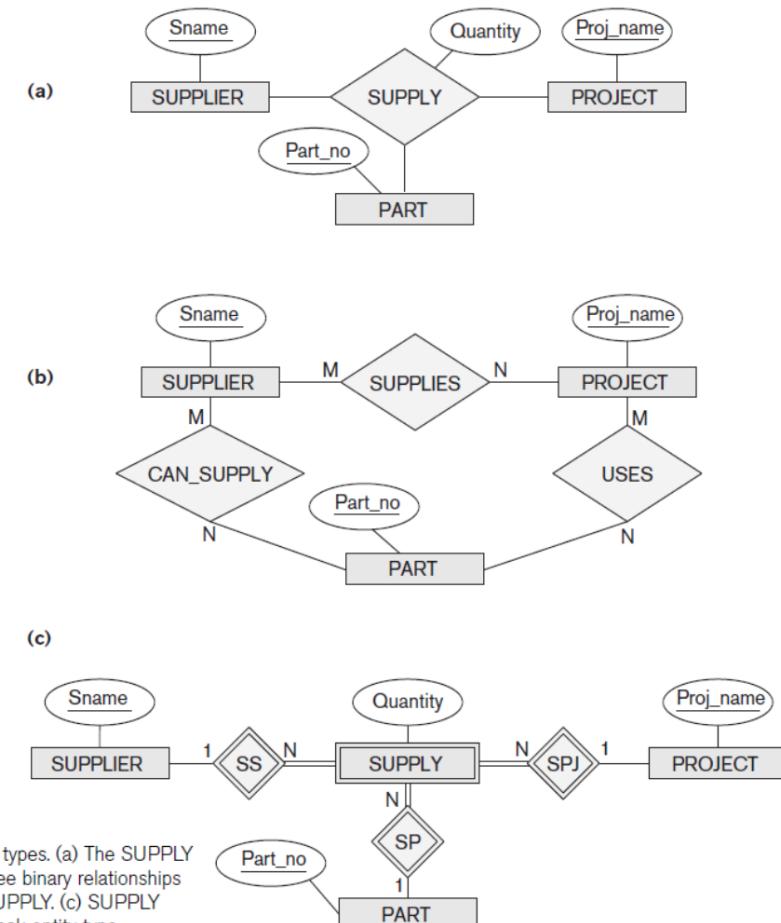


Figure 7.17

Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

9 ER to Relational

1. Erős egyedtípusok leképezése

- Az ER séma minden E erős egyedtípusához rendeljünk hozzá egy R relációsémát, amely tartalmazza E összes egyszerű attribútumát. Az összetett attribútumoknak csak az egyszerű komponenseit adjuk hozzá R attribútumaihoz.
- Válasszuk E kulcs attribútumainak egyikét az R relációséma elsődleges kulcsául. Ha az E-ből választott kulcs összetett, akkor annak egyszerű attribútumai együttesen fogják alkotni R elsődleges kulcsát.
- R egyedreláció

2. Gyenge egyedtípusok leképezése

- Az ER séma minden W gyenge egyedtípusához rendeljünk hozzá egy R relációsémát, melynek attribútumai legyenek W összes egyszerű attribútuma és W összetett attribútumainak egyszerű komponensei. Továbbá adjuk hozzá R attribútumaihoz külső kulcs attribútumként azoknak a tulajdonos relációsémáknak az elsődleges kulcs attribútumait.

- R elsődleges kulcsa a tulajdonos egyedtípusok elsődleges kulcsainak és a W gyenge egyedtípus diszkriminátorának az együttese.

3. Bináris 1 : 1 számosságú kapcsolattípusok leképezése

- Külső kulcs használata: Válasszuk ki az egyik relációt (mondjuk S-t) és vegyük fel S külső kulcsaként T elsődleges kulcsát. Célszerű S-nek azt a relációt választani, amelyiket abból az egyedtípusból képeztünk le, amelyik totális résztvevője az R kapcsolatnak.
- Vegyük fel továbbá R egyszerű attribútumait, illetve R összetett attribútumainak egyszerű komponenseit S attribútumaiként.

vagy

- Összevonás: Egy másik lehetőség az 1 : 1 kapcsolatok leképezésére, ha a két egyedtípust és a kapcsolatot egyetlen relációba vonjuk össze. Ezt akkor tehetjük meg, ha minden két egyedtípus totális résztvevő a kapcsolatnak.

vagy

- Kereszthivatkozás vagy kapcsoló reláció használata: A harmadik lehetőség, hogy felveszünk egy harmadik R relációt abból a célból, hogy kereszthivatkozás-sal lássuk el a két egyedtípusból képzett S és T relációk elsődleges kulcsait.
- Az R reláció az S és a T elsődleges kulcsait tartalmazza, mint külső kulcs. Az R elsődleges kulcsa S vagy T elsődleges kulcsa lesz, a másik külső kulcsra pedig egyediségi megszorítást tehetünk.
- Vegyük fel továbbá R egyszerű attribútumait, illetve R összetett attribútumainak egyszerű komponenseit S attribútumaiként.
- Az R relációt kapcsoló relációnak nevezzük

4. Bináris 1 : N számosságú kapcsolattípusok leképezése

- R kapcsolattípus esetén meg kell határozni azt az S relációt, amelyiket a kapcsolattípus N-oldali egyedtípusából képeztünk.
- Vegyük fel S külső kulcsaként az R-ben részt vevő másik egyedtípusból képzett T reláció elsődleges kulcsát.
- Vegyük fel továbbá R egyszerű attribútumait, illetve R összetett attribútumainak egyszerű komponenseit S attribútumaiként.

vagy

- most is használhatunk kapcsoló relációt (kereszthivatkozást), ahogy az 1 : 1 kapcsolatoknál tettük. Ekkor egy külön R relációt hozunk létre, amelynek attribútumai S és T elsődleges kulcsai, és amelynek elsődleges kulcsa megegyezik S elsődleges kulcsával.

5. Bináris M : N számosságú kapcsolattípusok leképezése

- hozzunk létre egy új S relációt, amely R-et reprezentálja. Vegyük fel S külső kulcsaként a kapcsolatban részt vevő egyedtípusokból képzett relációk elsődleges kulcsait; ezek együttese alkotja S elsődleges kulcsát. Vegyük fel továbbá R

egyszerű attribútumait, illetve R összetett attribútumainak egyszerű komponenseit S attribútumaiként.

- S kapcsoló reláció

6. Többértékű attribútumok leképezése

- hozzunk létre egy új R relációt.
- Ez az R reláció tartalmazza egy, az A-nak megfelelő attribútumot, valamint annak a relációnak a K elsődleges kulcsát - R külső kulcsaként -, amelyet az A-t tartalmazó egyedtípusból vagy kapcsolattípusból képeztünk. R elsődleges kulcsát A és K együttese alkotja.
- Ha a többértékű attribútum összetett, akkor az egyszerű komponenseit vegyük fel R attribútumaiként.

7. N-edfokú kapcsolattípusok leképezése

- hozzunk létre egy új S relációt, amely R-et reprezentálja.
- Vegyük fel S külső kulcsaként a kapcsolatban részt vevő egyedtípusokból képzett relációk elsődleges kulcsait. Vegyük fel továbbá R egyszerű attribútumait, illetve R összetett attribútumainak egyszerű komponenseit S attribútumaiként.
- S elsődleges kulcsa általában az összes külső kulcs együttese.

10 EER Model

EER modell minden ER modellbeli fogalmat tartalmaz, és azokon felül még tartalmazza a következő fogalmakat:

- Alosztály és szuperosztály
- Specializáció és generalizáció
- Kategória vagy uniótípus
- Attribútum és kapcsolat öröklődés

10.1 Alosztály, szuperosztály, és öröklődés

Egy egyedtípus altípusa vagy alosztálya egyedei egy alcsoporthoz, amelynek van jelentősége.

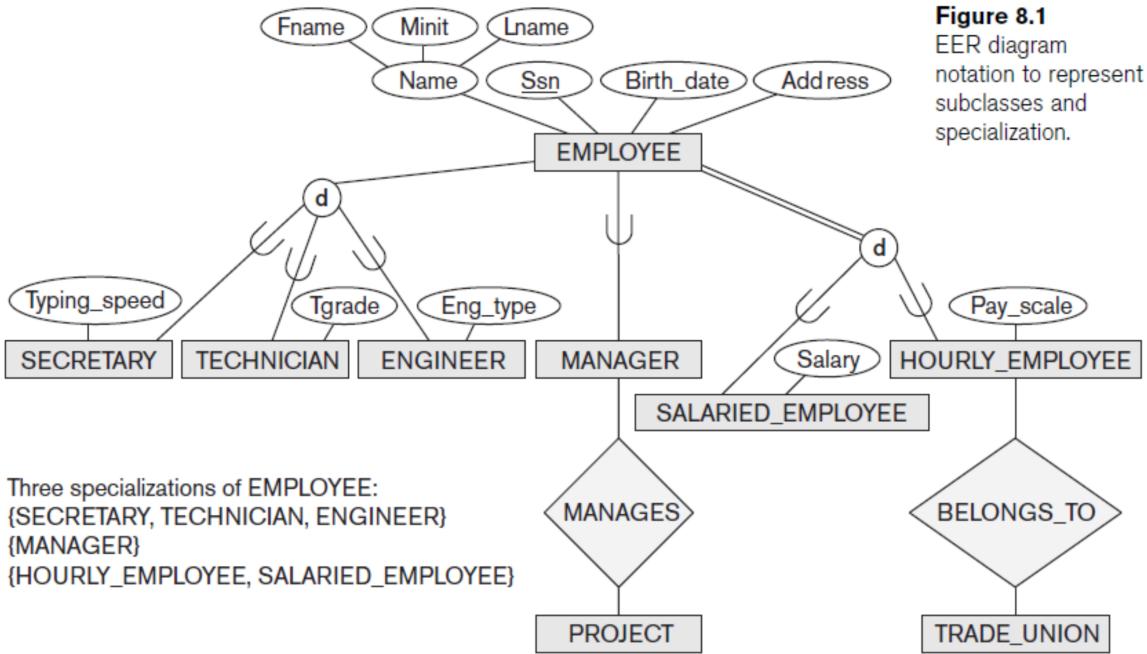
Az adatbázis-alkalmazásra vonatkozó jelentőségek miatt külön kell őket reprezentálni

Például:

Secretary, Technician, Enginner, Manager - alosztályai az Employee szuperosztálynak

10.1.1 Típus öröklődés

Az alosztályok egyedei öröklik a szuperosztály minden attribútumát és kapcsolatát



10.2 Specializáció

Az a folyamat, amely egy egyedtípus alosztályait definiálja.

Az alosztályok definiálása a szuperosztályban lévő egyedek bizonyos jellemzői alapján történik.

Az EER diagramon egy specializációs körrel jelölik.

Az alosztályokat lehet meghatározni:

- Attribútum(ok) alapján
- Vagy kapcsolattípus alapján

Példa:

Secretary, Technician, stb. - alosztályai az Employee szuperosztálynak, a meghatározó attribútum a job.

Hourly_employee alosztálya az Employee szuperosztálynak, a meghatározó kapcsolattípus a Belongs_to

10.2.1 Generalizáció

Az a folyamat, amelyben adott egyedtípusokból egy általános egyedtípust határozunk meg.

A specializáció ellentétes folyamata

Általánosítás egy szuperosztályba: Az eredeti egyedtípusok speciális alosztályok

Lehet egy vagy több alosztály

Egyed altípus meghatározása:

- **Predikátumdefiniált** (vagy feltételdefiniált) alosztályok (predikátum: job_type='Secretary') (az EER diagramon az alosztály és a spec. kör közötti vonal mellé írjuk)
- **Attribútumdefiniált** specializáció (az EER diagramon a spec. kör mellé írjuk az attribútum nevét, az értékét az alosztály mellé)
- Felhasználó által definiált (a felhasználó az egyes egyedekről egyesével eldönti, hogy melyik alosztályba tartoznak)

10.2.2 A specializáció és generalizáció megszorításai

Diszjunkt megszorítás

Meghatározza, hogy a specializáció altípusai diszjunktak (azaz egy egyed csak egy altípus eleme lehet).

A spec. körbe d-t írunk.

Ha az alosztályok nem diszjunktak, akkor az egyedeik halmaza átfedő, o-val jelöljük a körben.

Teljesség megszorítás

Lehet teljes vagy részleges

Teljes specializáció: a szupertípus minden egyedének legalább egy alosztályban szerepelnie kell (dupla vonal)

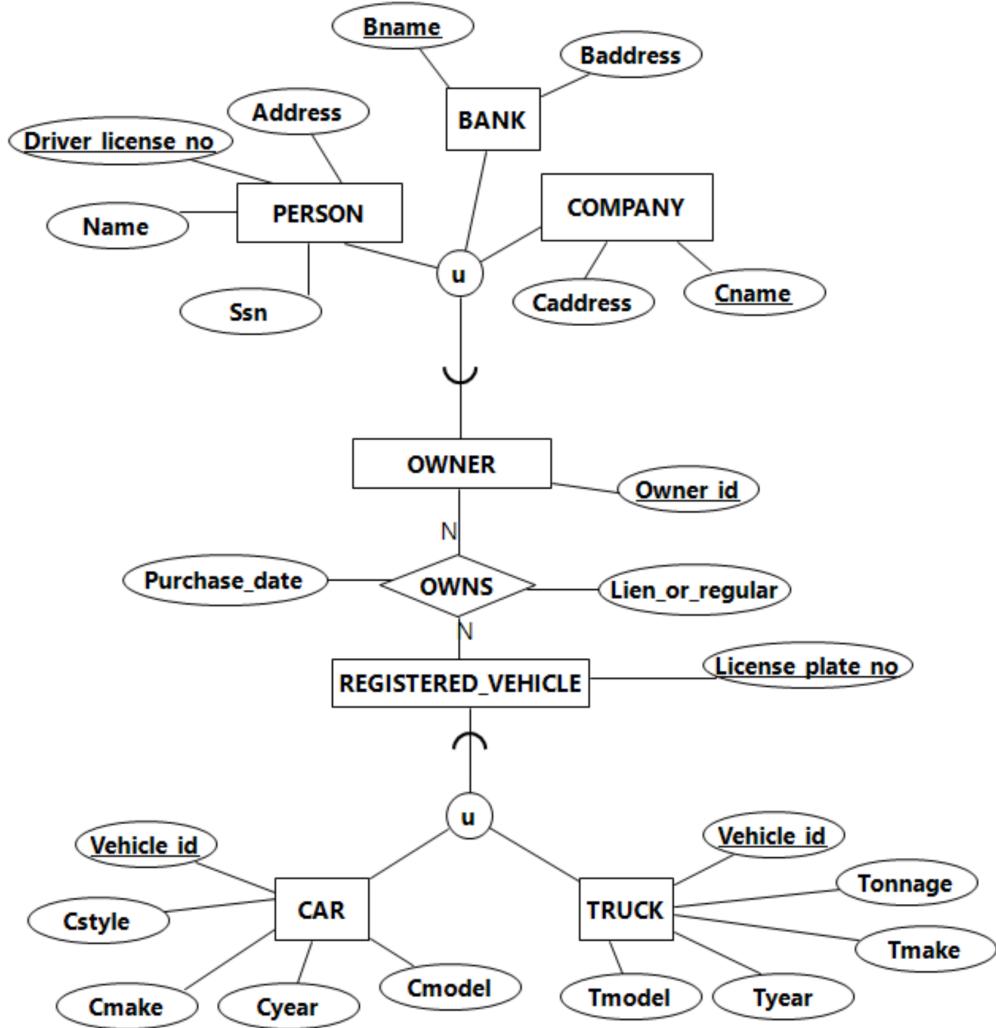
Részleges specializáció: enged olyan egyedeiket, amelyek egyik alosztályhoz sem tartoznak.

A diszjunkság és a teljesség független egymástól

10.2.3 Uniótípus modellezése kategóriák használatával

Uniótípus vagy kategória

- Egy szuperosztály/atosztály kapcsolatot képvisel, ahol egynél több szuperosztály van
- Az alosztály reprezentálja az objektumok kollekcióját, amely a különböző szuperosztályok uniójának részhalmaza
- Az attribútumöröklődés sokkal szelektívebben működik
- A kategória lehet totális vagy részleges
- A spec. körben egy u jelöli



11 EER modell leképezése relációs modellre

- Specializációk és generalizációk leképezésének lehetőségei
 - 8A: Több reláció - szuperosztály és alosztályok
 - Ez a lehetőség mindenféle specializáció esetén (totális vagy részleges, diszjunkt vagy átfedő) működik.
 - Hozzunk létre egy relációt a C szuperosztály számára C attribútumaival, az elsődleges kulcsa k legyen.
 - Hozzunk létre egy relációt minden alosztályhoz S_i ($1 \leq i \leq m$) az S_i attribútumaival és k-val. Az elsődleges kulcs k.
 - 8B: Több reláció - csak alosztály relációk
 - Ez a lehetőség csak olyan specializáció esetén működik, ahol az alosztályok totálisak. Ha a specializáció átfedő, egy egyed több relációban is felbukkanhat, emiatt diszjunkt alosztályoknál javasolt.

- Hozzunk létre egy S_i relációt minden alosztályhoz az S_i és a C attribútumaival. Az elsődleges kulcs a k.
 - 8C: Egyetlen reláció egy típusattribútummal
 - Ez a lehetőség csak olyan specializáció esetén működik, amely diszjunkt.
 - Fennáll a veszélye annak, hogy ez a megoldás sok NULL értéket generál, ha sok speciális attribútum szerepel az alosztályban.
 - Hozzunk létre egyetlen relációt a következő attribútumokkal:

$$\{k, a_1, \dots, a_n\} \cup \{S_1\text{attribútumai}\} \cup \dots \cup \{S_m\text{attribútumai}\} \cup \{t\}$$
 Az elsődleges kulcs a k
 - A t-t típus (vagy diszkrimináló) attribútumnak nevezzük, amely jelzi azt az alosztályt, amelyhez az egyes rekordok tartoznak.
 - 8D: Egyetlen reláció több típusattribútummal
 - Ez a lehetőség olyan specializációk esetén is működik, amely átfedő alosztályokat tartalmaz.
 - Diszjunkt specializációra is megfelelő
 - Hozzunk létre egy relációt a következő attribútumokkal:

$$\{k, a_1, \dots, a_n\} \cup \{S_1\text{attribútumai}\} \cup \dots \cup \{S_m\text{attribútumai}\} \cup \{t_1, t_2, \dots, t_m\}$$
 - . Az elsődleges kulcs k.
 - minden $t_i (1 \leq i \leq m)$ egy logikai típusú attribútum, amely azt jelzi, hogy egy adott rekord az S_i osztályhoz tartozik-e.
2. Kategóriák (uniótípusok) leképezése
- Különböző kulcsokkal rendelkező szuperosztályok által definiált kategória leképezéséhez célszerű egy új kulcsattribútumot bevezetni, amelyet helyettesítő kulcsnak nevezünk a kategóriának megfelelő reláció létrehozásakor.
 - A helyettesítő kulcs attribútumot minden olyan relációba is felvesszük, amelyeket a kategória szuperosztályaiból képezünk.

12 Normal Forms

12.1 A funkcionális függés definíciója

Az R relációséma két attribútumhalmaza, X és Y között, $X \rightarrow Y$ -nal jelölt funkcionális függés előír egy megszorítást azokra a lehetséges rekordokra, amelyek az R egy r relációs állapotát valósíthatják meg. A megszorítás az, hogy bármely két, r -beli t_1 és t_2 rekord esetén, amelyekre $t_1[X] = t_2[X]$ teljesül, teljesülnie kell $t_1[Y] = t_2[Y]$ -nak is.

Az attribútumok szemantikájának vagy jelentésének a tulajdonsága

Minden lehetséges relációs állapotnak meg kell felelnie a funkcionális függés megszorításnak

12.2 A funkcionális függés tulajdonságai

- **Reflexív:** $X \supseteq Y \Rightarrow X \rightarrow Y$
- **Augmentív:** $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$
- **Tranzitív:** $X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$
- **Dekompozíciós tulajdonság:**
$$X \rightarrow YZ \Rightarrow X \rightarrow Y$$
- **Additív:** $X \rightarrow Y, X \rightarrow Z \Rightarrow X \rightarrow YZ$
- **Pszeudotranzitív:** $X \rightarrow Y, WY \rightarrow Z \Rightarrow WX \rightarrow Z$

12.3 Lezárt

F: egy R relációs sémán meghatározott funkcionális függések halmaza.

F lezártja funkcionális függések halmaza, amely minden olyan funkcionális függést tartalmaz, amely F-ból következtethető.

Jele: F^+

12.4 Armstrong axióma

A funkcionális függés reflexív, augmentív és tranzitív szabálya együtt helyes és teljes.

Helyes: ha adott egy R relációsémán fennálló funkcionális függéseknek egy F halmaza, akkor bármilyen függés, amely levezethető F-ből a három szabály segítségével, fenn fog állni R minden olyan r relációjában, amely kielégíti az F-beli függéseket.

Teljes: az F^+ (F lezártja) halmaz meghatározható F-ből a 3 szabály alkalmazásával.

A reflexivitás, az augmentivitás és a tranzitivitás szabályait együtt Armstrong-axiómáknak nevezzük.

12.5 Első Normálforma

A relációs modellbeli reláció formális definíciójának a része

Csak atomi (vagy oszthatatlan) értékek szerepelhetnek az 1NF-ben (azaz nem szerepelhetnek összetett és halmazértékű attribútumok)

Feltételezzük, hogy a relációt van elsődleges kulcsa.

Hogyan érjük el az első normálformát?

- Távolítsuk el az attribútumot egy másik relációba
- (Bővítsük a kulcsot)
- Használunk több atomi attribútumot

12.6 Részleges függés

Egy $X \rightarrow Y$ funkcionális függés részleges függés, ha X valamely A attribútumait eltávolítva X-ből a függés még fennáll.

Egy $X \rightarrow Y$ funkcionális függés teljes funkcionális függés, ha X bármely A attribútumát eltávolítva X-ből a függés már nem áll fenn.

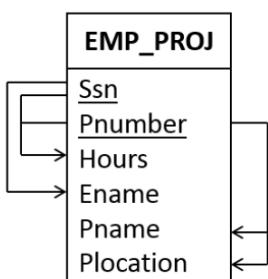
12.7 Második Normálforma

Egy R relációséma második normálformában (2NF-ben) van, ha első normálformában van és R minden nem elsőrendű (leíró) attribútuma teljesen funkcionálisan függ R elsődleges kulcsától (azaz nem tartalmaz részleges függést).

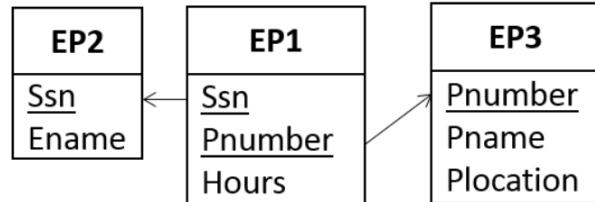
2NF-re alakítás:

- Az eredeti relációból eltávolítjuk a részegeSEN függő nem elsőrendű (leíró) attribútumot (A) egy másik relációba. Ebben a második relációban szerepelnie kell az eredeti reláció elsődleges kulcsának azon részének (B), amelytől a nem elsőrendű attribútum függ. A második reláció elsődleges kulcsa B, azaz az eredeti reláció elsődleges kulcsának része.

1NF, de nem 2NF



2NF



12.8 Tranzitív függés

Egy R relációséma $X \rightarrow Y$ funkcionális függése tranzitív függés, ha létezik egy olyan Z attribútumhalmaz R-ben, amely nem kulcsjelölt és nem része R egyetlen kulcsának sem, és fennáll $X \rightarrow Z$ és $Z \rightarrow Y$.

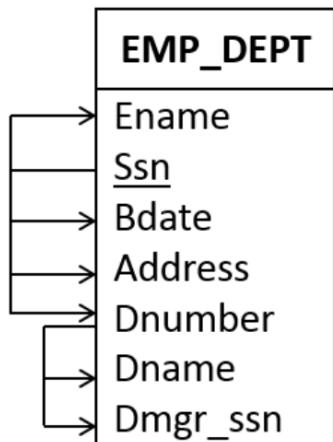
12.9 Harmadik Normálforma

Egy R relációséma harmadik normálformában (3NF-ben) van, ha második normálformában (2NF-ben) van, és nincs R-nek olyan nem elsőrendű (leíró) attribútuma, amely tranzitívan függne az elsődleges kulcstól.

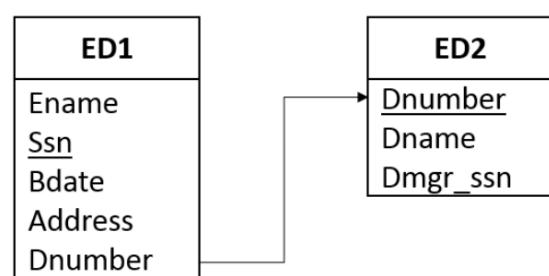
3NF-re alakítás:

Az eredeti relációból eltávolítjuk a tranzitívan függő nem elsőrendű (leíró) attribútumot egy másik relációba. Ebben a második relációban elsődleges kulcsként kell szerepelnie azoknak az attribútumoknak, amelyektől a nem elsőrendű attribútumok függenek.

2NF, de nem 3NF



3NF



12.10 Boyce-Codd Normálforma

Egy R relációséma Boyce-Codd-féle normálformában (BCNF-ben) van, ha valahányszor egy $X \rightarrow A$ nemtriviális funkcionális függés fennáll R-en, akkor X egy szuperkulcsa R-nek.

Minden BCNF-ben lévő reláció 3NF-ben is van, viszont a 3NF-ben lévő relációk nem szükségszerűen vannak BCNF-ben

3NF-ben A lehet elsőrangú, BCNF-ben nem

12.11 Többértékű függés

Az 1NF következménye (kettő vagy több halmazértékű attribútum esetén kibővíjtük az elsődleges kulcsot)

Egy R relációsémán megadott $X \rightarrow Y$ többértékű függés (ahol X és Y R attribútumhalmazai) a következő megszorítást jelenti R bármely r relációs állapotára vonatkozóan:

Ha van két olyan t_1 és t_2 rekord r-ben, amelyre $t_1[X] = t_2[X]$, akkor léteznie kell két t_3 és t_4 rekordnak is r-ben a következő tulajdonságokkal, ahol Z-t az $(R - (X \cup Y))$ jelölésére használjuk:

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ és $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ és $t_4[Z] = t_1[Z]$.

$(X \twoheadrightarrow Y: X$ többértékűen meghatározza Y -t)

12.12 Negyedik normálforma (4NF)

Feltételei sérülnek, ha egy relációban nemkívánatos többértékű függés van.

Egy R relációséma negyedik normálformában (4NF-ben) van, figyelembe véve az F függések halmazát (amely magában foglalja a funkcionális és többértékű függéseket), ha minden F^+ -beli nemtriviális $X \rightarrow\!\!\! \rightarrow Y$ többértékű függés esetén X szuperkulcsa R-nek.

Ha egy reláció egy nemtriviális többértékű függés miatt nincs 4NF-ben, akkor bontsuk fel 4NF-ben lévő relációk egy halmazára.

— —

Emp_projects

| <u>Ename</u> | <u>Pname</u> |
|--------------|--------------|
| Smith | X |
| Smith | Y |

Eredeti:

Emp

| <u>Ename</u> | <u>Pname</u> | <u>Dname</u> |
|--------------|--------------|--------------|
| Smith | X | John |
| Smith | Y | Anna |
| Smith | X | Anna |
| Smith | Y | John |

Emp_dependents

| <u>Ename</u> | <u>Dname</u> |
|--------------|--------------|
| Smith | John |
| Smith | Anna |

13 UML

13.1 Interaction models

All systems involve interaction of some kind.

- user interaction, which involves user inputs and outputs: it helps to identify user requirements.
- interaction between the system being developed and other systems,
- interaction between the components of the system.

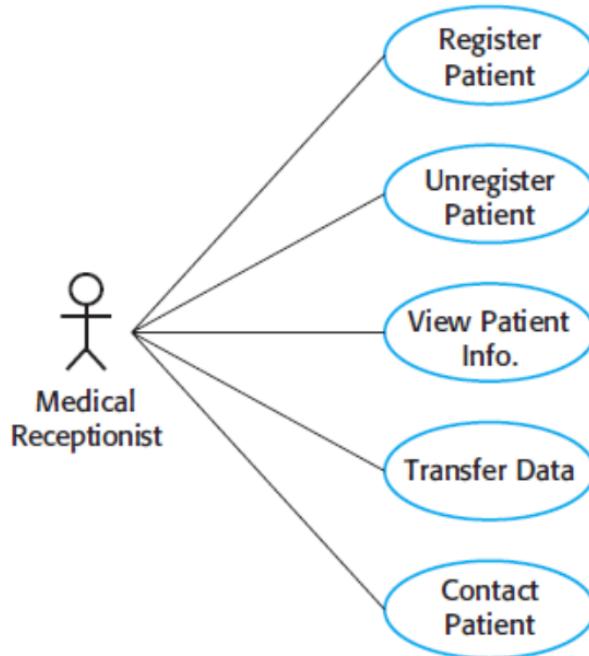
13.1.1 Use case modeling

Use case models and sequence diagrams present interaction at different levels of detail and so may be used together.

A use case can be taken as a simple scenario that describes what a user expects from a system.

Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures.

Use case diagrams give a fairly simple overview of an interaction so you have to provide more detail to understand what is involved. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram.

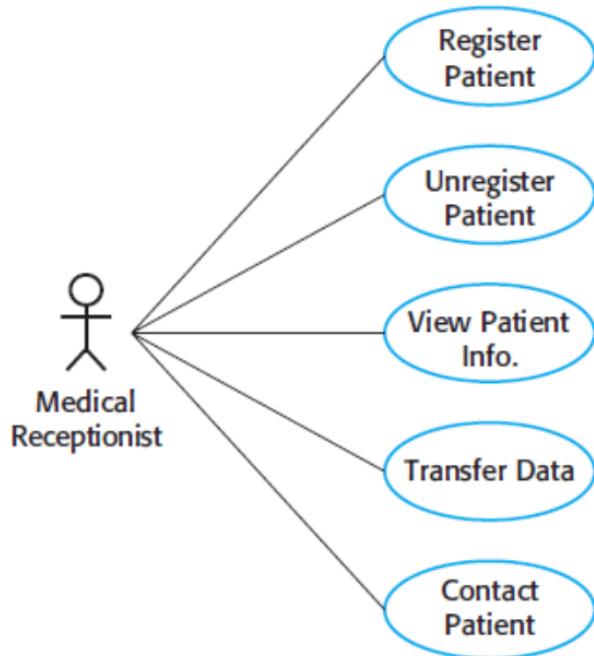


13.1.2 Sequence diagrams

Primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves.

The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled.

It shows the sequence of interactions that take place during a particular use case or use case instance.



- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.
- The rectangle on the dotted lines indicates the lifeline of the object concerned.
- You read the sequence of interactions from top to bottom.
- The annotations on the arrows indicate the calls to the objects, their parameters, and the return values.

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships.

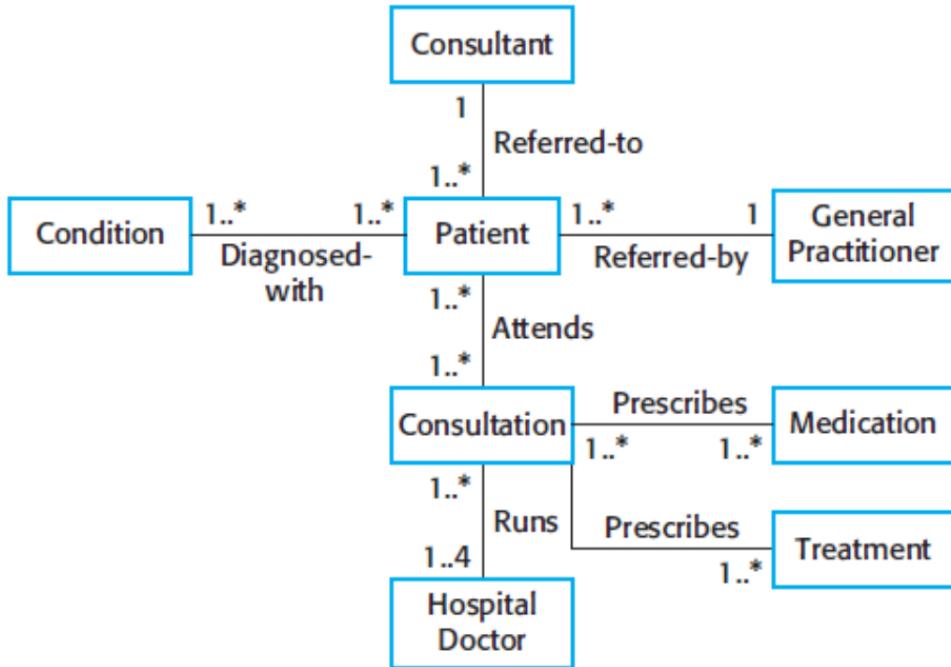
13.1.3 Class Diagrams

Class diagrams are used when developing an object- oriented system model to show the classes in a system and the associations between these classes.

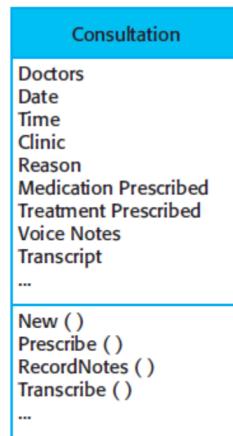
An object class can be thought of as a general definition of one kind of system object.

An association is a link between classes that indicates that there is a relationship between these classes.

Each class may have to have some knowledge of its associated class.

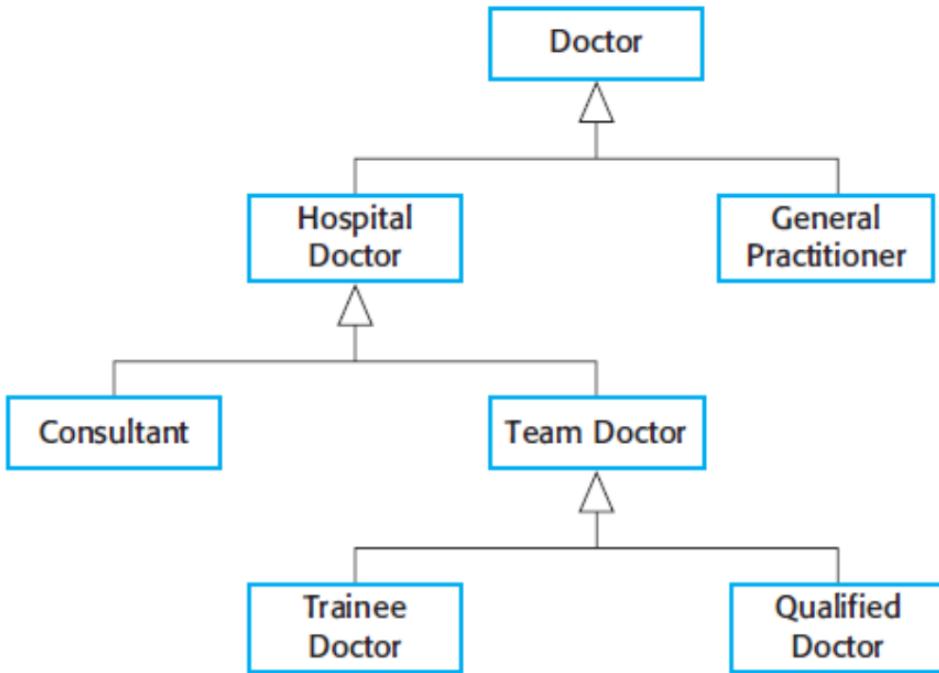


At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design. They show the data entities, their associated attributes, and the relations between these entities.



In the UML, you show attributes and operations by extending the simple rectangle that represents a class:

- The name of the object class is in the top section.
- The class attributes are in the middle section. This must include the attribute names and, optionally, their types.
- The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle.



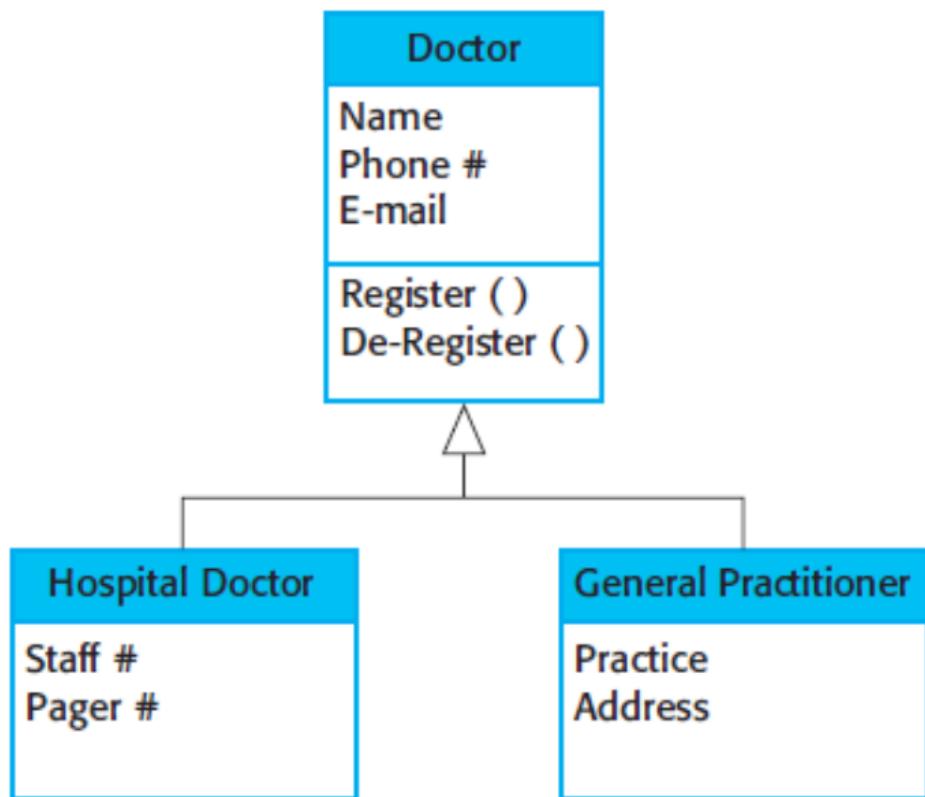
Generalization is used to manage complexity. Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of these classes. This allows us to infer that different members of these classes have some common characteristics (e.g., squirrels and rats are rodents). We can make general statements that apply to all class members (e.g., all rodents have teeth for gnawing).

In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization.

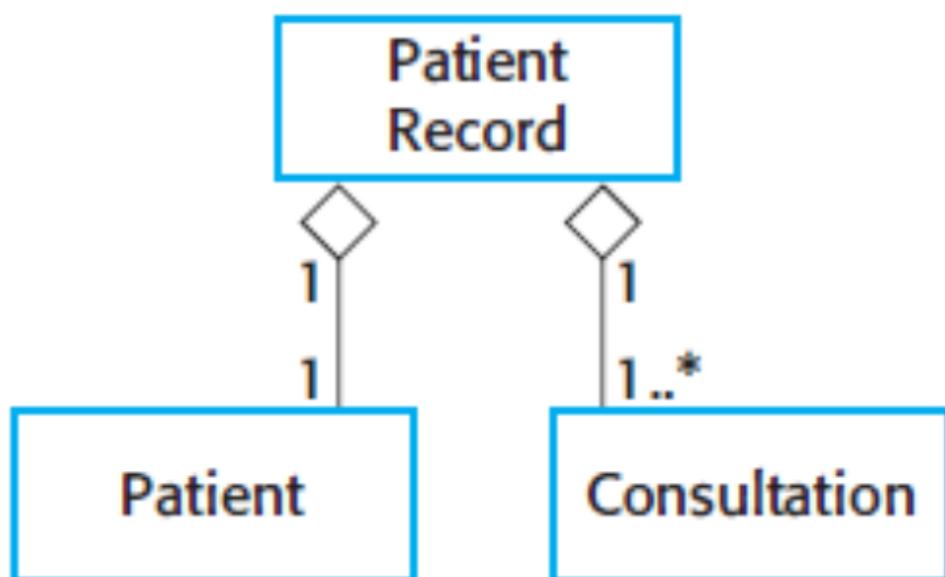
This means that common information will be maintained in one place only.

If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.

In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.



Aggregation



Objects in the real world are often composed of different parts.

Aggregation means that one object (the whole) is composed of other objects (the parts).

To show this, we use a diamond shape next to the class that represents the whole.

13.1.4 Behavioral models

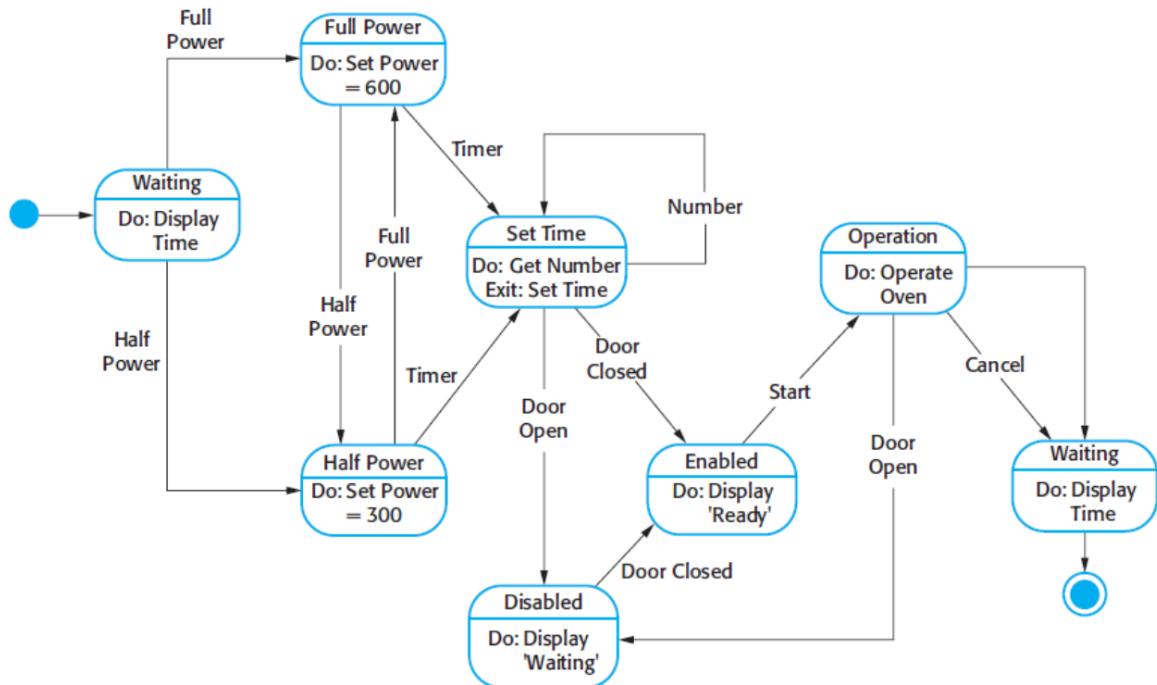
Event-driven modeling

It shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

It is particularly appropriate for real-time systems.

The UML supports event-based modeling using state diagrams, which were based on Statecharts. State diagrams show system states and events that cause transitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

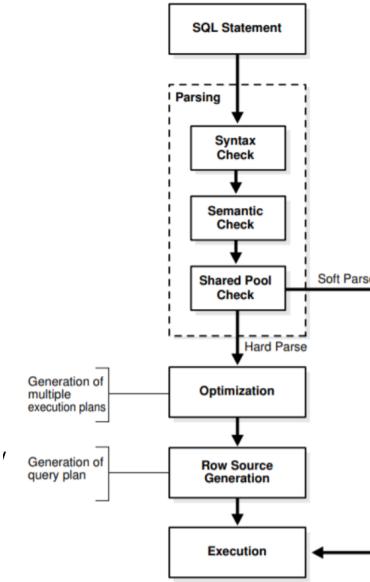
In UML state diagrams, rounded rectangles represent system states. They may include a brief description (following 'do') of the actions taken in that state. The labeled arrows represent stimuli that force a transition from one state to another. You can indicate start and end states using filled circles, as in activity diagrams.



For large system models, you need to hide detail in the models. One way to do this is by using the notion of a superstate that encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram.

14 Query Optimization

14.1 SQL Processing



14.1.1 SQL Parsing

The parsing stage involves separating the pieces of a SQL statement into a data structure that other routines can process. The database parses a statement when instructed by the application, which means that only the application, and not the database itself, can reduce the number of parses. When an application issues a SQL statement, the application makes a parse call to the database to prepare the statement for execution. The parse call opens or creates a cursor, which is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information. The cursor and private SQL area are in the program global area (PGA). During the parse call, the database performs checks that identify the errors that can be found before statement execution. Some errors cannot be caught by parsing. For example, the database can encounter deadlocks or errors in data conversion only during statement execution.

Syntax Check

Oracle Database must check each SQL statement for syntactic validity. A statement that breaks a rule for well-formed SQL syntax fails the check.

Semantic Check

The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist. A syntactically correct statement can fail a semantic check (nonexistent_table):
SQL> SELECT * FROM nonexistent_table;

Shared Pool Check

During the parse, the database performs a shared pool check to determine whether it can skip resource-intensive steps of statement processing. To this end, the database uses a

hashing algorithm to generate a hash value for every SQL statement. The statement hash value is the SQL ID shown in V\$SQL.SQL_ID. When a user submits a SQL statement, the database searches the shared SQL area to see if an existing parsed statement has the same hash value.

Parse operations fall into the following categories, depending on the type of statement submitted and the result of the hash check:

- **Hard parse:** If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code. This operation is known as a hard parse, or a library cache miss. During the hard parse, the database accesses the library cache and data dictionary cache numerous times to check the data dictionary.
- **Soft parse:** A soft parse is any parse that is not a hard parse. If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses.

14.1.2 SQL Optimization

During optimization, Oracle Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse. The database does not optimize DDL. The only exception is when the DDL includes a DML component such as a subquery that requires optimization.

14.1.3 SQL Row Source Generation

The row source generator is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database.

The iterative plan is a binary program that, when executed by the SQL engine, produces the result set. The plan takes the form of a combination of steps. Each step returns a row set. The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement.

A **row source** is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows. The row source can be a table, view, or result of a join or grouping operation. The row source generator produces a row source tree, which is a collection of row sources. The row source tree shows the following information:

- An ordering of the tables referenced by the statement
- An access method for each table mentioned in the statement
- A join method for tables affected by join operations in the statement
- Data operations such as filter, sort, or aggregation

The **execution plan** for the statement is the output of the row source generator.

14.1.4 SQL Execution

During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing.

In general, the order of the steps in execution is the reverse of the order in the plan, so you read the plan from the bottom up. Each step in an execution plan has an ID number. The numbers correspond to the Id column in the plan. Initial spaces in the Operation column of the plan indicate hierarchical relationships. For example, if the name of an operation is preceded by two spaces, then this operation is a child of an operation preceded by one space. Operations preceded by one space are children of the SELECT statement itself.

The SQL engine executes the plan as follows:

- Step 6 uses a full table scan to retrieve all rows from the departments table.
- Step 5 uses a full table scan to retrieve all rows from the jobs table.
- Step 4 scans the emp_name_ix index in order, looking for each key that begins with the letter A and retrieving the corresponding rowid.
- Step 3 retrieves from the employees table the rows whose rowids were returned by Step 4.
- Step 2 performs a hash join, accepting row sources from Steps 3 and 5, joining each row from the Step 5 row source to its corresponding row in Step 3, and returning the resulting rows to Step 1.
- Step 1 performs another hash join, accepting row sources from Steps 2 and 6, joining each row from the Step 6 source to its corresponding row in Step 2, and returning the result to the client.

During execution, the database reads the data from disk into **memory** if the data is not in memory. The database also takes out any **locks** and latches necessary to ensure data integrity and logs any changes made during the SQL execution. The final stage of processing a SQL statement is **closing the cursor**.

14.2 Query Optimizer Concepts

The query optimizer (called simply the optimizer) is built-in database software that determines the most efficient method for a SQL statement to access requested data. Purpose of the Query Optimizer: The optimizer attempts to generate the most optimal execution plan for a SQL statement. The optimizer choose the plan with the lowest cost among all considered candidate plans. The optimizer uses available statistics to calculate cost. For a specific query in a given environment, the cost computation accounts for factors of query execution such as I/O, CPU, and communication. For example, a query might request information about employees who are managers. If the optimizer statistics indicate that 80% of employees are managers, then the optimizer may decide that a full table scan is most efficient. However, if statistics indicate that very few employees are managers, then reading an index followed by a table access by rowid may be more efficient than a full table scan. Because the database has many internal statistics and tools at its disposal, the optimizer is usually in a better position than the user to determine the optimal method of statement execution. For this reason, all SQL statements use the optimizer.

14.2.1 Cost-Based Optimization

Query optimization is the process of choosing the most efficient means of executing a SQL statement. SQL is a nonprocedural language, so the optimizer is free to merge, reorganize, and process in any order. The database optimizes each SQL statement based on statistics collected about the accessed data. The optimizer determines the optimal plan for a SQL statement by examining multiple access methods, such as full table scan or index scans, different join methods such as nested loops and hash joins, different join orders, and possible transformations. For a given query and environment, the optimizer assigns a relative numerical cost to each step of a possible plan, and then factors these values together to generate an overall cost estimate for the plan. After calculating the costs of alternative plans, the optimizer chooses the plan with the lowest cost estimate. For this reason, the optimizer is sometimes called the cost-based optimizer (CBO) to contrast it with the legacy rule-based optimizer (RBO).

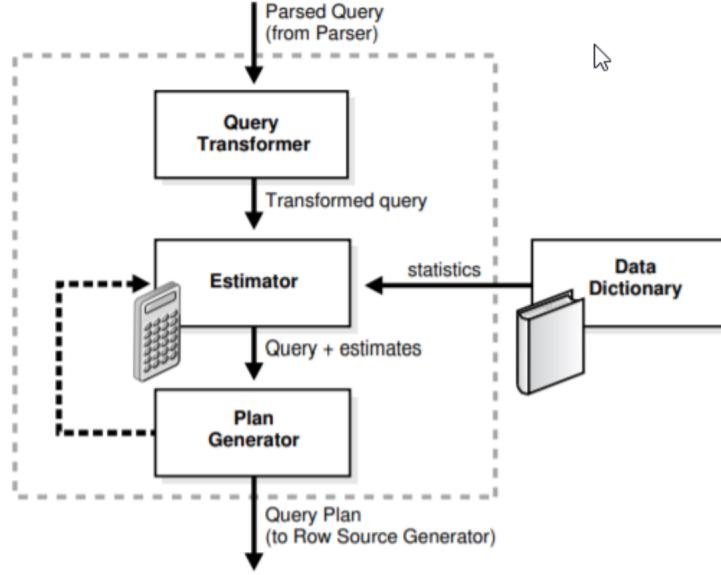
14.2.2 Execution Plans

An execution plan describes a **recommended method of execution for a SQL statement**. The plan shows the **combination of the steps** Oracle Database uses to execute a SQL statement. Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement. An execution plan **displays the cost** of the entire plan, indicated on line 0, and each separate operation. **The cost is an internal unit that the execution plan only displays to allow for plan comparisons.** Thus, you cannot tune or change the cost value.

Query Subplans

For each query block, the optimizer generates a query subplan. The database optimizes query blocks separately from the bottom up. Thus, the database optimizes the innermost query block first and generates a subplan for it, and then generates the outer query block representing the entire query. The number of possible plans for a query block is proportional to the number of objects in the FROM clause. This number rises exponentially with the number of objects. For example, the possible plans for a join of five tables are significantly higher than the possible plans for a join of two tables.

14.3 Optimizer Components



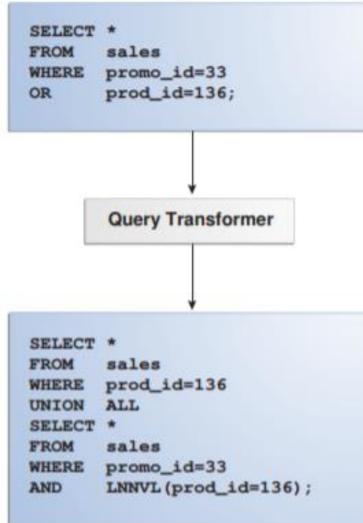
- **Query Transformer:** The optimizer determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan.
- **Estimator:** The optimizer estimates the cost of each plan based on statistics in the data dictionary.
- **Plan Generator:** The optimizer compares the costs of plans and chooses the lowest-cost plan, known as the execution plan, to pass to the row source generator.

14.3.1 Query Transformer

For some statements, the query transformer determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost. When a viable alternative exists, the database calculates the cost of the alternatives separately and chooses the lowest- cost alternative.

14.3.2 Query Transformer

An example: the query transformer rewriting an input query that uses OR into an output query that uses UNION ALL.



14.3.3 Estimator

The estimator is the component of the optimizer that determines the overall cost of a given execution plan. The estimator uses three different measures to determine cost:

- **Selectivity:** The percentage of rows in the row set that the query selects, with 0 meaning no rows and 1 meaning all rows. Selectivity is tied to a query predicate, such as WHERE last_name LIKE 'A%', or a combination of predicates. A predicate becomes more selective as the selectivity value approaches 0 and less selective (or more unselective) as the value approaches 1.
- **Cardinality:** The cardinality is the number of rows returned by each operation in an execution plan. This input, which is crucial to obtaining an optimal plan, is common to all cost functions. The estimator can derive cardinality from the table statistics collected by DBMS_STATS, or derive it after accounting for effects from predicates (filter, join, and so on), DISTINCT or GROUP BY operations, and so on. The Rows column in an execution plan shows the estimated cardinality.
- **Cost:** This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

If statistics are available, then the estimator uses them to compute the measures. The statistics improve the degree of accuracy of the measures.

14.3.4 Plan Generator

The plan generator explores various plans for a query block by trying out different access paths, join methods, and join orders. Many plans are possible because of the various combinations that the database can use to produce the same result. The optimizer picks the plan with the lowest cost. The optimizer uses an internal cutoff to reduce the number of plans it tries when finding the lowest-cost plan. The cutoff is based on the cost of the current best plan. If the current best cost is large, then the optimizer explores alternative plans to find a lower cost plan. If the current best cost is small, then the optimizer ends the search swiftly because further cost improvement is not significant.

14.4 Query Transformations

It describes the most important optimizer techniques for transforming queries. The optimizer decides whether to use an available transformation based on cost. Transformations may not be available to the optimizer for a variety of reasons, including hints or lack of constraints. For example, transformations such as subquery unnesting are not available for hybrid partitioned tables, which contain external partitions that do not support constraints.

14.4.1 OR Expansion

In OR expansion, the optimizer transforms a query block containing top-level disjunctions into the form of a UNION ALL query that contains two or more branches. The optimizer achieves this goal by splitting the disjunction into its components, and then associating each component with a branch of a UNION ALL query. The optimizer can choose OR expansion for various reasons. For example, it may enable more efficient access paths or alternative join methods that avoid Cartesian products. As always, the optimizer performs the expansion only if the cost of the transformed statement is lower than the cost of the original statement. In previous releases, the optimizer used the CONCATENATION operator to perform the OR expansion. Starting in Oracle Database 12c Release 2 (12.2), the optimizer uses the UNION-ALL operator instead.

```
SELECT * FROM employees e, departments d WHERE (e.email='SSTILES' OR d.department_name= AND e.department_id = d.department_id;
```

```
SELECT * FROM employees e, departments d WHERE e.email = 'SSTILES' AND e.department_id = d.department_id UNION ALL SELECT * FROM employees e, departments d WHERE d.department_name = 'Treasury' AND e.department_id = d.department_id;
```

14.4.2 View Merging

In view merging, the optimizer merges the query block representing a view into the query block that contains it. View merging can improve plans by enabling the optimizer to consider additional join orders, access methods, and other transformations. For example, after a view has been merged and several tables reside in one query block, a table inside a view may permit the optimizer to use join elimination to remove a table outside the view. For certain simple views in which merging always leads to a better plan, the optimizer automatically merges the view without considering cost. Otherwise, the optimizer uses cost to make the determination. The optimizer may choose not to merge a view for many reasons, including cost or validity restrictions (privileges). You can use hints to override view merging rejected because of cost

In **simple view merging**, the optimizer merges select-project-join views. For example, a query of the employees table contains a subquery that joins the departments and locations tables.

Simple view merging frequently results in a more optimal plan because of the additional join orders and access paths available after the merge.

A view may not be valid for simple view merging because:

- The view contains constructs not included in select-project-join views, including: GROUP BY, DISTINCT, Outer join, MODEL, CONNECT BY, Set operators, Aggregation
- The view appears on the right side of a semijoin (Exists, in) or antijoin (not exist, not in).
- The view contains subqueries in the SELECT list.
- The outer query block contains PL/SQL functions.
- The view participates in an outer join, and does not meet one of the several additional validity requirements that determine whether the view can be merged.

The following query joins the `hr.employees` table with the `dept_locs_v` view, which returns the street address for each department. `dept_locs_v` is a join of the `departments` and `locations` tables.

```
SELECT e.first_name, e.last_name, dept_locs_v.street_address, dept_locs_v.postal_code
FROM employees e, (SELECT d.department_id, d.department_name, l.street_address, l.postal_code
                    FROM departments d, locations l
                   WHERE d.location_id = l.location_id ) dept_locs_v
WHERE dept_locs_v.department_id = e.department_id AND e.last_name = 'Smith';
```

The database can execute the preceding query by joining `departments` and `locations` to generate the rows of the view, and then joining this result to `employees`. Because the query contains the view `dept_locs_v`, and this view contains two tables, the optimizer must use one of the following join orders:

- `employees, dept_locs_v` (departments, locations)
- `employees, dept_locs_v` (locations, departments)
- `dept_locs_v` (departments, locations), `employees`
- `dept_locs_v` (locations, departments), `employees`

Join methods are also constrained. The index-based nested loops join is not feasible for join orders that begin with `employees` because no index exists on the column from this view. Without view merging, the optimizer generates the following execution plan:

Complex View Merging

The transformed query is cheaper than the untransformed query, so the optimizer chooses to merge the view. In the untransformed query, the **GROUP BY operator applies to the entire sales table in the view**. In the transformed query, the joins to products and customers **filter out a large portion of the rows** from the sales table, so the **GROUP BY operation is lower cost**. The join is more expensive because the sales table has not been reduced, but it is not much more expensive because the GROUP BY operation does not reduce the size of the row set very much in the original query. If any of the preceding characteristics were to change, merging the view might no longer be lower cost.

The following query of the `cust_prod_v` view uses a DISTINCT operator:

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name, c.cust_email
```

```

FROM customers c, products p,
(SELECT DISTINCT s.cust_id, s.prod_id FROM sales s) cust_prod_v
WHERE c.country_id = 52790 AND c.cust_id = cust_prod_v.cust_id
AND cust_prod_v.prod_id = p.prod_id
AND p.prod_name = 'T3 Faux Fur-Trimmed Sweater';

```

After determining that view merging produces a lower-cost plan, the optimizer rewrites the query into this equivalent query:

```

SELECT nwv.cust_id, nwv.cust_first_name, nwv.cust_last_name, nwv.cust_email
FROM ( SELECT DISTINCT(c.rowid), p.rowid, s.prod_id, s.cust_id,
c.cust_first_name, c.cust_last_name, c.cust_email
FROM customers c, products p, sales s
WHERE c.country_id = 52790 AND c.cust_id = s.cust_id
AND s.prod_id = p.prod_id
AND p.prod_name = 'T3 Faux Fur-Trimmed Sweater' ) nwv;

```

14.4.3 Predicate Pusing

In predicate pushing, the optimizer "pushes" the relevant predicates from the containing query block into the view query block. For views that are not merged, this technique improves the subplan of the unmerged view. The database can use the pushed-in predicates to access indexes or to use as filters.

14.4.4 Join Factorization

In the cost-based transformation known as **join factorization**, the optimizer can factorize common computations from branches of a UNION ALL query.

UNION ALL queries are common in database applications, especially in data integration applications. Often, branches in a UNION ALL query refer to the same base tables. Without join factorization, the optimizer evaluates each branch of a UNION ALL query independently, which leads to repetitive processing, including data access and joins. Join factorization transformation can share common computations across the UNION ALL branches. Avoiding an extra scan of a large base table can lead to a huge performance improvement.

Join factorization can factorize multiple tables and from more than two UNION ALL branches.

```

SELECT t1.c1, t2.c2
FROM t1, t2, t3
WHERE t1.c1 = t2.c1
AND t1.c1 > 1
AND t2.c2 = 2
AND t2.c2 = t3.c2
UNION ALL
SELECT t1.c1, t2.c2
FROM t1, t2, t4
WHERE t1.c1 = t2.c1
AND t1.c1 > 1
AND t2.c3 = t4.c3

```

Without any transformation, the database must perform the scan and the filtering on table t1 twice, one time for each branch.

```
SELECT t1.c1, VW_JF_1.item_2
FROM t1, (SELECT t2.c1 item_1, t2.c2 item_2
FROM t2, t3
WHERE t2.c2 = t3.c2
AND t2.c2 = 2
UNION ALL
SELECT t2.c1 item_1, t2.c2 item_2
FROM t2, t4
WHERE t2.c3 = t4.c3) VW_JF_1
WHERE t1.c1 = VW_JF_1.item_1 AND t1.c1 > 1
```

In this case, because table t1 is factorized, the database performs the table scan and the filtering on t1 only one time. If t1 is large, then this factorization avoids the huge performance cost of scanning and filtering t1 twice.

14.5 SQL Operators: Access Paths and Joins

A row source is a set of rows returned by a step in the execution plan. A SQL operator acts on a row source. A unary operator acts on one input, as with access paths. A binary operator acts on two outputs, as with joins. An access path is a technique used by a query to retrieve rows from a row source.

A row source is a set of rows returned by a step in an execution plan. A row source can be a table, view, or result of a join or grouping operation. A unary operation such as an access path, which is a technique used by a query to retrieve rows from a row source, accepts a single row source as input. For example, a full table scan is the retrieval of rows of a single row source. In contrast, a join is binary and receives inputs from exactly two row sources. The database uses different access paths for different relational data structures. The following table summarizes common access paths for the major data structures.

In general, index access paths are more efficient for statements that retrieve a small subset of table rows, whereas full table scans are more efficient when accessing a large portion of a table.

14.5.1 Table Access Paths

A table is the basic unit of data organization in an Oracle database. Relational tables are the most common table type. Relational tables have with the following organizational characteristics:

- A heap-organized table does not store rows in any particular order.
- An index-organized table orders rows according to the primary key values.
- An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.

Heap-Organized Table Access

By default, a table is organized as a heap, which means that the database places rows where they fit best rather than in a user-specified order. As users add rows, the database places the rows in the first available free space in the data segment. Rows are not guaranteed to be retrieved in the order in which they were inserted. Every row in a heap-organized table has a rowid unique to this table that corresponds to the physical address of a row piece. A rowid is a 10-byte physical address of a row. Oracle Database uses rowids internally for the construction of indexes.

A rowid is an internal representation of the storage location of data. The rowid of a row specifies the data file and data block containing the row and the location of the row in that block. Locating a row by specifying its rowid is the fastest way to retrieve a single row because it specifies the exact location of the row in the database. In most cases, the database accesses a table by rowid after a scan of one or more indexes. However, table access by rowid need not follow every index scan. If the index contains all needed columns, then access by rowid might not occur.

A **sample table scan** retrieves a random sample of data from a simple table or a complex SELECT statement, such as a statement involving joins and views. The database uses a sample table scan when a statement FROM clause includes the SAMPLE keyword.

B-Tree Index Access Paths

An index is an optional structure, associated with a table or table cluster, that can sometimes speed data access. By creating an index on one or more columns of a table, you gain the ability in some cases to retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of reducing disk I/O. B-trees, short for balanced trees, are the most common type of database index. A B-tree index is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.

D-4: Index Access Paths

Index Unique Scans: An index unique scan returns at most 1 **rowid**.

Index Range Scans: An index range scan is an ordered scan of values. The optimizer typically chooses a range scan for queries with **high selectivity**. By default, the database stores indexes in ascending order, and scans them in the same order. An **index range scan descending** is identical to an index range scan except that the database returns rows in descending order.

Index Full Scans: An index full scan reads the entire index in order. An index full scan can eliminate a separate sorting operation because the data in the index is ordered by index key.

Bitmap Index Access Paths

Bitmap indexes combine the indexed data with a rowid range. In a conventional B-tree index, one index entry points to a single row. In a bitmap index, the key is the combination of the indexed data and the rowid range. The database stores at least one bitmap for each index key. Each value in the bitmap, which is a series of 1 and 0 values, points to

a row within a rowid range. Thus, in a bitmap index, one index entry points to a set of rows rather than a single row.

- Bitmap Index Single Value: This type of access path uses a bitmap index to look up a single key value.
- Bitmap Index Range Scans: This type of access path uses a bitmap index to look up a range of values.
- Bitmap Merge: This access path merges multiple bitmaps, and returns a single bitmap as a result. A bitmap merge is indicated by the BITMAP MERGE operation in an execution plan.

Table Cluster Access Paths

A table cluster is a group of tables that share common columns and store related data in the same blocks. When tables are clustered, a single data block can contain rows from multiple tables. Cluster Scans: An index cluster is a table cluster that uses an index to locate data Hash Scans: A hash cluster is like an indexed cluster, except the index key is replaced with a hash function. No separate cluster index exists. In a hash cluster, the data is the index. The database uses a hash scan to locate rows in a hash cluster based on a hash value.

Joins

A join combines the output from exactly two row sources, such as tables or views, and returns one row source. The returned row source is the data set. Whenever multiple tables exist in the FROM clause, Oracle Database performs a join. A join condition compares two row sources using an expression. The join condition defines the relationship between the tables. If the statement does not specify a join condition, then the database performs a Cartesian join, matching every row in one table with every row in the other table.

The following query joins the `hr.employees` table with the `dept_locs_v` view, which returns the street address for each department. `dept_locs_v` is a join of the departments and locations tables.

```
SELECT e.first_name, e.last_name, dept_locs_v.street_address, dept_locs_v.postal_code
FROM employees e, (SELECT d.department_id, d.department_name, l.street_address, l.postal_code
                    FROM departments d, locations l
                   WHERE d.location_id = l.location_id ) dept_locs_v
WHERE dept_locs_v.department_id = e.department_id AND e.last_name = 'Smith';
```

The database can execute the preceding query by joining departments and locations to generate the rows of the view, and then joining this result to employees. Because the query contains the view `dept_locs_v`, and this view contains two tables, the optimizer must use one of the following join orders:

- employees, `dept_locs_v` (departments, locations)
- employees, `dept_locs_v` (locations, departments)
- `dept_locs_v` (departments, locations), employees
- `dept_locs_v` (locations, departments), employees

Join methods are also constrained. The index-based nested loops join is not feasible for join orders that begin with employees because no index exists on the column from this view. Without view merging, the optimizer generates the following execution plan:

- **Nested Loops Joins:** Nested loops join an outer data set to an inner data set. For each row in the outer data set that matches the single-table predicates, the database retrieves all rows in the inner data set that satisfy the join predicate. If an index is available, then the database can use it to access the inner data set by rowid.
- **Hash Joins:** The database uses a hash join to join larger data sets. The optimizer uses the smaller of two data sets to **build a hash table on the join key in memory**, using a deterministic hash function to specify the location in the hash table in which to store each row. The database then scans the larger data set, probing the hash table to find the rows that meet the join condition
- **Sort Merge Joins:** A sort merge join is a variation on a nested loops join. If the two data sets in the join are not already sorted, then the database **sorts** them. These are the SORT JOIN operations. For each row in the first data set, the database probes the second data set for matching rows and joins them, basing its start position on the match made in the previous iteration.

14.6 Optimizer Controls

15 Software development

There are two kinds of software products:

- **Generic products:** These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them.
- **Customized (or bespoke) products:** These are systems that are commissioned by a particular customer. A software contractor develops the software especially for that customer.

A **software process** is a sequence of activities that leads to the production of a software product.

Four fundamental activities that are common to all software processes:

- **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
- **Software development**, where the software is designed and programmed.
- **Software validation**, where the software is checked to ensure that it is what the customer requires.
- **Software evolution**, where the software is modified to reflect changing customer and market requirements.

15.1 Software process models

- **The waterfall model:** This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases.
- **Incremental development:** This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
- **Reuse-oriented software engineering:** This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

15.1.1 The waterfall model

Plan-driven process: you must plan and schedule all of the process activities before starting work on them.

1. *Requirements analysis and definition:* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design:* The systems design process allocates the requirements to either hardware or software systems by establishing an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. *Implementation and unit testing:* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.
4. *Integration and system testing:* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance:* Normally (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase is one or more documents that are approved ('signed off').

The following phase should not start until the previous phase has finished. In practice, these stages overlap and feed information to each other.

Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework.

The problems of the model may cause that the system won't do what the user wants.

It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

15.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed.

Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.

It is a fundamental part of agile approaches

It is better than a waterfall approach for most business, e-commerce, and personal systems.

Incremental development reflects the way that we solve problems: We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake.

By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer.

Generally, the early increments of the system include the most important or most urgently required functionality.

This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three important benefits, compared to the waterfall model:

- The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much less than is required with the waterfall model.
- It is easier to get customer feedback on the development work that has been done.
- More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

The incremental approach has two problems:

- The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system.

Large systems need a **stable framework or architecture** and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

15.1.3 Reuse-oriented software engineering

People working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as word processing or a spreadsheet.

Stages:

1. *Requirements specification*
2. *Component analysis*: Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.
3. *Requirements modification*: During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.
4. *System design with reuse*: During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.
5. *Development and integration*: Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.
6. *Validation*

There are three types of software component that may be used in a reuse-oriented process:

- Web services that are developed according to service standards and which are available for remote invocation.
- Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
- Stand-alone software systems that are configured for use in a particular environment.

15.2 Process activities

The activities in the design process vary, depending on the type of system being developed. The four activities that may be part of the design process for information systems:

1. *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called sub-systems or modules), their relationships, and how they are distributed.
2. *Interface design*, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component can be used without other components having to know how it is implemented. Once interface specifications are agreed, the components can be designed and developed concurrently.
3. *Component design*, where you take each system component and design how it will operate. This may be a simple statement of the expected functionality to be implemented, with the specific design left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model. The design model may be used to automatically generate an implementation.
4. *Database design*, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.

16 Architectural Design

You can design software architectures at two levels of abstraction:

- **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

System architectures are often modeled using simple block diagrams.

Each box in the diagram represents a component.

Boxes within boxes indicate that the component has been decomposed to sub-components.

Arrows mean that data and or control signals are passed from component to component in the direction of the arrows.

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand.

There are two ways in which an architectural model of a program is used:

- As a way of facilitating discussion about the system design

- As a way of documenting an architecture that has been designed

Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

- Performance
- Security
- Safety (event of failure)
- Availability
- Maintainability

It is suggested that there should be four fundamental architectural views:

- **A logical view:** which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.
- **A process view:** which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about nonfunctional system characteristics such as performance and availability.
- **A development view:** which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
- **A physical view:** which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

16.1 Architectural Patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used.

You can think of an architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments.

So, an architectural pattern should describe a system organization that has been successful in previous systems.

It should include information of when it is and is not appropriate to use that pattern, and the pattern's strengths and weaknesses.

16.1.1 Layered Architecture

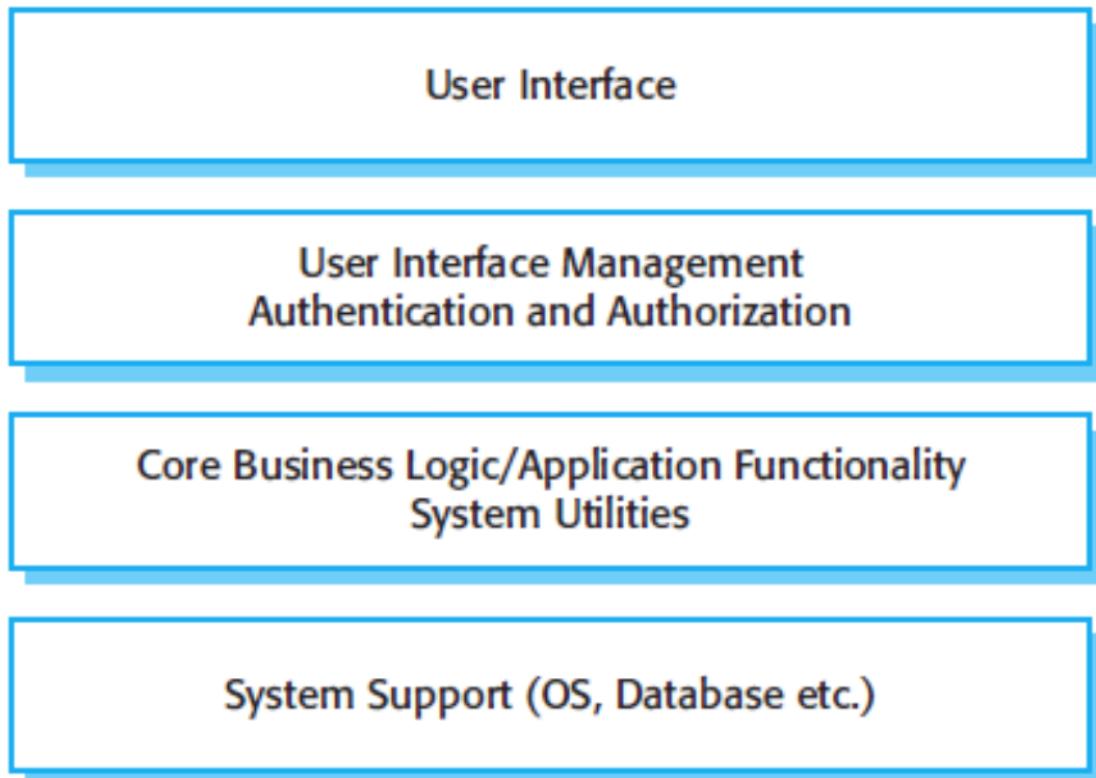
The system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

This layered approach supports the incremental development of systems.

As a layer is developed, some of the services provided by that layer may be made available to users.

The architecture is changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer.

When layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected.



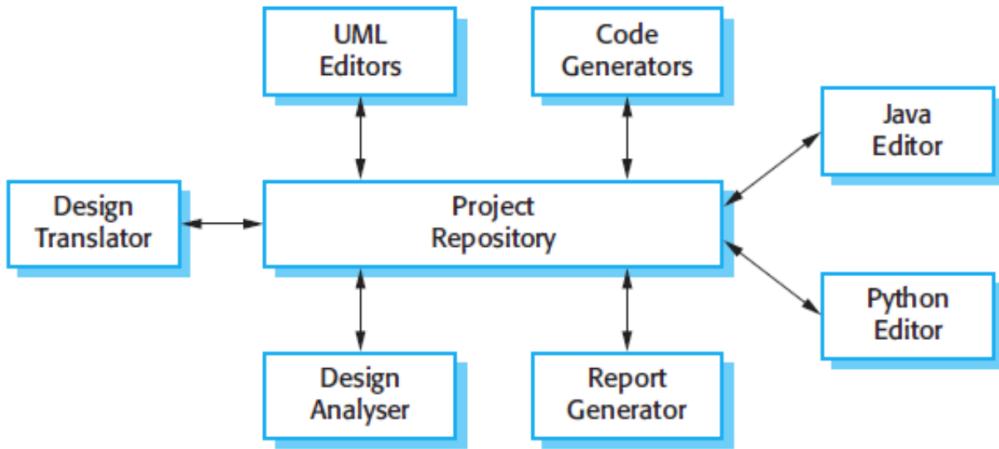
- **Description:** Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.
- **Example:** A layered model of a system for sharing copyright documents held in different libraries.
- **When used:** Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
- **Advantages:** Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
- **Disadvantages:** In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower- level layers

rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

16.1.2 Repository Architecture

It describes how a set of interacting components can share data.

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another.



- **Description:** All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
- **Example:** command and control systems, management information systems, CAD systems, and interactive development environments for software.
- **When used:** You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
- **Advantages:** Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
- **Disadvantages:** The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

16.1.3 Client-server Architecture

A system that follows the client-server pattern is organized as a set of services and associated servers, and clients that access and use the services.

The major components of this model are:

- A set of servers that offer services to other components.
- A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
- A network that allows the clients to access these services. Most client-server systems are implemented as distributed systems, connected using Internet protocols.
- **Description:** The functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
- **Example:** A film and video/DVD library organized as a client-server system.
- **When used:** Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
- **Advantages:** The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
- **Disadvantages:** Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

16.1.4 Pipe and filter architecture (Event-Driven)

This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs.

Data flows from one to another and is transformed as it moves through the sequence.

Each processing step is implemented as a transform.

Input data flows through these transforms until converted to output.

The transformations may execute sequentially or in parallel.

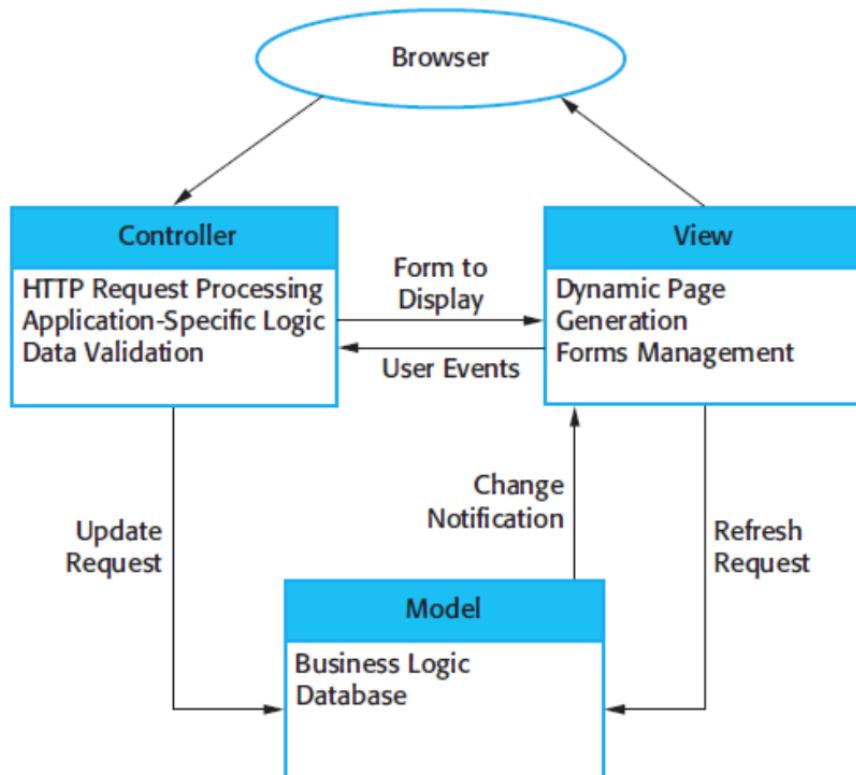
The data can be processed by each transform item by item or in a single batch.

- **Description:** The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
- **Example:** A billing system used for processing invoices.

- **When used:** Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
- **Advantages:** Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
- **Disadvantages:** The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

16.1.5 MVC Pattern

This pattern is the basis of interaction management in many web-based systems.



- **Description:** Separates presentation and interaction from the system data. The system is structured into
- three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.

- **Example:** The architecture of a web-based application system organized using the MVC pattern.
- **When used:** Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
- **Advantages:** Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
- **Disadvantages:** Can involve additional code and code complexity when the data model and interactions are simple.