**1.2. Baze de date relaționale**

A **relational database** is a digital database whose organization is based on the relational model of data, as proposed by E.F. Codd in 1970. This model organizes data into one or more tables (or "relations") of rows and columns, with a unique key for each row. Generally, each entity type described in a database has its own table, the rows representing instances of that type of entity and the columns representing values attributed to that instance. Because each row in a table has its own unique key, rows in a table can be linked to rows in other tables by storing the unique key of the row to which it should be linked (where such unique key is known as a "foreign key"). Codd showed that data relationships of arbitrary complexity can be represented using this simple set of concepts.

Prior to the advent of this model, databases were usually hierarchical, and each tended to be organized with a unique mix of indexes, chains, and pointers. The simplicity of the relational model led to it soon becoming the predominant type of database.

The various software systems used to maintain relational databases are known as Relational Database Management Systems (RDBMS).

Virtually all relational database systems use SQL (Structured Query Language) as the language for querying and maintaining the database.

**1.2.1. Prezentare generală**

Each database is a collection of tables, which are called relations, hence the name "relational database". Each table is a metaphorical representation of an entity or object that is in a tabular format consisting of columns and rows. Columns are the fields of a record or the attributes of an entity. The rows contain the values or data instances; these are also called records or tuples.

Relationships exist both among the columns within a table and among the tables. These relationships take three logical forms: one-to-one, one-to-many, or many-to-many. Most relational databases are designed so there is only one value per cell (an intersection of a column and row); in this design pattern, there are only one-to-one relationships within a table. Each table is named according to the data it contains, such as people or addresses.

In order for a database management system (DBMS) to operate efficiently and accurately, it must have ACID transactions. Part of this processing involves consistently being able to select or modify one and only one row in a table. Therefore, most physical implementations have a system-assigned, unique primary key for each table. When a new row is written to the table, the system generates and writes the new, unique value for the primary key (PK); this is the key that the system uses primarily for accessing the table. System performance is optimized for PKs. Other, more natural keys may also be identified and defined as alternate keys (AK). Often several columns may be needed to form an AK (this is one reason why a single integer column is usually made the PK). Both PKs and AKs have the ability to uniquely identify one row within a table. Additional technology may be applied that will significantly assure a unique ID across the world, a globally unique identifier; these are used when there are broader system requirements.

The primary keys within a database are used to define the relationships among the tables. When a PK migrates to another table, it becomes a foreign key in the other table. When each cell can contain only one value and the PK migrates into a regular entity table, this design pattern can represent either a one-to-one, or a one-to-many relationship. Most relational database designs resolve many-to-many relationships by creating an additional table that contains the PKs from both of the other entity tables—the relationship becomes an entity; the resolution table is then named appropriately and is often assigned its own PK while the two FKs are combined to form an AK. The migration of PKs to other tables is the second major reason why system-assigned integers are used normally as PKs; there usually is neither efficiency nor clarity in migrating a bunch of other types of columns.

Most of the programming within a RDBMS is accomplished using stored procedures (SPs). Often procedures can be used to greatly reduce the amount of information transferred within and outside of a system. For increased security, the system design may also grant access to only the stored procedures and not directly to the tables. Fundamental stored procedures contain the logic needed to insert new data and update existing data. More complex procedures may be written to implement additional rules and logic related to processing or selecting the data.

**1.2.2. Terminologii**

|  |  |  |
| --- | --- | --- |
| SQL term | Relational database term | Description |
| Row | Tuple or record | A data set representing a single item |
| Column | Attribute or field | A labeled element of a tuple, e.g. "Address" or "Date of birth" |
| Table | Relation or Base relation variable | A set of tuples sharing the same attributes; a set of columns and rows |
| View or result set | Derived relation variable | Any set of tuples; a data report from the RDBMS in response to a query |

Tabel 1.1

**1.2.3. Relații sau tabele**

A *relation* is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints.

The relational model specifies that the tuples of a relation have no specific order and that the tuples, in turn, impose no order on the attributes. Applications access data by specifying queries, which use operations such as *select* to identify tuples, *project* to identify attributes, and *join* to combine relations. Relations can be modified using the *insert*, *delete*, and *update* operators. New tuples can supply explicit values or be derived from a query. Similarly, queries identify tuples for updating or deleting.

Tuples by definition are unique. If the tuple contains a candidate or primary key then obviously it is unique; however, a primary key need not be defined for a row or record to be a tuple. The definition of a tuple requires that it be unique, but does not require a primary key to be defined. Because a tuple is unique, its attributes by definition constitute a superkey.

**1.2.4. Relații de baza și derivate**

In a relational database, all data are stored and accessed via relations. Relations that store data are called "base relations", and in implementations are called "tables". Other relations do not store data, but are computed by applying relational operations to other relations. These relations are sometimes called "derived relations". In implementations these are called "views" or "queries". Derived relations are convenient in that they act as a single relation, even though they may grab information from several relations. Also, derived relations can be used as an abstraction layer.

**1.2.5. Domenii**

A domain describes the set of possible values for a given attribute, and can be considered a constraint on the value of the attribute. Mathematically, attaching a domain to an attribute means that any value for the attribute must be an element of the specified set. The character string *"ABC"*, for instance, is not in the integer domain, but the integer value*123* is. Another example of domain describes the possible values for the field "Gender" as ("Male,"Female"). So, the field "Gender" will not accept input values like (0.1) or (M,F).

**1.2.6. Constrângeri**

A **primary key** uniquely specifies a tuple within a table. In order for an attribute to be a good primary key it must not repeat. While natural attributes (attributes used to describe the data being entered) are sometimes good primary keys, surrogate keys are often used instead. A surrogate key is an artificial attribute assigned to an object which uniquely identifies it (for instance, in a table of information about students at a school they might all be assigned a student ID in order to differentiate them). The surrogate key has no intrinsic (inherent) meaning, but rather is useful through its ability to uniquely identify a tuple. Another common occurrence, especially in regard to N:M cardinality is the composite key. A composite key is a key made up of two or more attributes within a table that (together) uniquely identify a record. (For example, in a database relating students, teachers, and classes. Classes *could* be uniquely identified by a composite key of their room number and time slot, since no other class could have exactly the same combination of attributes. In fact, use of a composite key such as this can be a form of data verification, albeit a weak one.

A **foreign key** is a field in a relational table that matches the primary key column of another table. The foreign key can be used to cross-reference tables. Foreign keys need not have unique values in the referencing relation. Foreign keys effectively use the values of attributes in the referenced relation to restrict the domain of one or more attributes in the referencing relation. A foreign key could be described formally as: "For all tuples in the referencing relation projected over the referencing attributes, there must exist a tuple in the referenced relation projected over those same attributes such that the values in each of the referencing attributes match the corresponding values in the referenced attributes."

A **stored procedure** is executable code that is associated with, and generally stored in, the database. Stored procedures usually collect and customize common operations, like inserting a tuple into a relation, gathering statistical information about usage patterns, or encapsulating complex business logic and calculations. Frequently they are used as an application programming interface (API) for security or simplicity. Implementations of stored procedures on SQL RDBMSs often allow developers to take advantage of procedural extensions (often vendor-specific) to the standard declarative SQL syntax. Stored procedures are not part of the relational database model, but all commercial implementations include them.

An **index** is one way of providing quicker access to data. Indices can be created on any combination of attributes on a relation. Queries that filter using those attributes can find matching tuples randomly using the index, without having to check each tuple in turn. This is analogous to using the index of a book to go directly to the page on which the information you are looking for is found, so that you do not have to read the entire book to find what you are looking for. Relational databases typically supply multiple indexing techniques, each of which is optimal for some combination of data distribution, relation size, and typical access pattern. Indices are usually implemented via B+ trees, R-trees, and bitmaps. Indices are usually not considered part of the database, as they are considered an implementation detail, though indices are usually maintained by the same group that maintains the other parts of the database. It should be noted that use of efficient indexes on both primary and foreign keys can dramatically improve query performance. This is because B-tree indexes result in query times proportional to log(n) where n is the number of rows in a table and hash indexes result in constant time queries (no size dependency as long as the relevant part of the index fits into memory).

**1.2.7. Operații relaționale**

Queries made against the relational database, and the derived relvars in the database are expressed in a relational calculus or a relational algebra. In his original relational algebra, Codd introduced eight relational operators in two groups of four operators each. The first four operators were based on the traditional mathematical set operations:

* The union operator combines the tuples of two relations and removes all duplicate tuples from the result. The relational union operator is equivalent to the SQL UNIONoperator.
* The intersection operator produces the set of tuples that two relations share in common. Intersection is implemented in SQL in the form of the INTERSECT operator.
* The difference operator acts on two relations and produces the set of tuples from the first relation that do not exist in the second relation. Difference is implemented in SQL in the form of the EXCEPT or MINUS operator.
* The cartesian product of two relations is a join that is not restricted by any criteria, resulting in every tuple of the first relation being matched with every tuple of the second relation. The cartesian product is implemented in SQL as the CROSS JOIN operator.

The remaining operators proposed by Codd involve special operations specific to relational databases:

* The selection, or restriction, operation retrieves tuples from a relation, limiting the results to only those that meet a specific criterion, i.e. a subset in terms of set theory. The SQL equivalent of selection is the SELECT query statement with a WHERE clause.
* The projection operation extracts only the specified attributes from a tuple or set of tuples.
* The join operation defined for relational databases is often referred to as a natural join. In this type of join, two relations are connected by their common attributes. MySQL's approximation of a natural join is the INNER JOIN operator. In SQL, an INNER JOIN prevents a cartesian product from occurring when there are two tables in a query. For each table added to a SQL Query, one additional INNER JOIN is added to prevent a cartesian product. Thus, for N tables in a SQL query, there must be N-1 INNER JOINS to prevent a cartesian product.
* The relational division operation is a slightly more complex operation, which involves essentially using the tuples of one relation (the dividend) to partition a second relation (the divisor). The relational division operator is effectively the opposite of the cartesian product operator (hence the name).

Other operators have been introduced or proposed since Codd's introduction of the original eight including relational comparison operators and extensions that offer support for nesting and hierarchical data, among others.

**1.3. Baze de date distribuite**

**1.3.1. Definiții**

We can define a distributed database (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a distributed database management system (DDBMS) as a software system that manages a distributed database while making the distribution transparent to the user.

For a database to be called distributed, the following minimum conditions should be satisfied:

- **Connection of database nodes over a computer network.** There are multiple computers, called **sites** or **nodes**. These sites must be connected by an underlying **communication network** to transmit data and commands among sites;

- **Logical interrelation of the connected databases.** It is essential that the information in the databases be logically related;

- **Absence of homogeneity constraint among connected nodes.** It is not necessary that all nodes be identical in terms of data, hardware, and software;

**1.3.2. Avantaje**

Some important advantages are listed below:

1. Improved ease and flexibility of application development. Developing and maintaining applications at geographically distributed sites of an organization is facilitated owing to transparency of data distribution and control.

2. Increased reliability and availability. This is achieved by the isolation of faults to their site of origin without affecting the other databases connected to the network. When the data and DDBMS software are distributed over several sites, one site may fail while other sites continue to operate. Only the data and software that exist at the failed site cannot be accessed. This improves both reliability and availability. Further improvement is achieved by judiciously replicating data and software at more than one site. In a centralized system, failure at a single site makes the whole system unavailable to all users. In a distributed database, some of the data may be unreachable, but users may still be able to access other parts of the database. If the data in the failed site had been replicated at another site prior to the failure, then the user will not be affected at all.

3. Improved performance. A distributed DBMS fragments the database by keeping the data closer to where it is needed most. Data localization reduces the contention for CPU and I/O services and simultaneously reduces access delays involved in wide area networks.When a large database is distributed over multiple sites, smaller databases exist at each site. As a result, local queries and transactions accessing data at a single site have better performance because of the smaller local databases. In addition, each site has a smaller number of transactions executing than if all transactions are submitted to a single centralized database. Moreover, interquery and intraquery parallelism can be achieved by executing multiple queries at different sites, or by breaking up a query into a number of subqueries that execute in parallel. This contributes to improved performance.

4. Easier expansion. In a distributed environment, expansion of the system in terms of adding more data, increasing database sizes, or adding more processors is much easier.

**1.3.3. Funcții adiționale**

Distribution leads to increased complexity in the system design and implementation. To achieve the potential advantages listed previously, the DDBMS software must be able to provide the following functions in addition to those of a centralized DBMS:

- **Keeping track of data distribution**. The ability to keep track of the data distribution, fragmentation, and replication by expanding the DDBMS catalog.

- **Distributed query processing**. The ability to access remote sites and transmit queries and data among the various sites via a communication network.

- **Distributed transaction management**. The ability to devise execution strategies for queries and transactions that access data from more than one site and to synchronize the access to distributed data and maintain the integrity of the overall database.

- **Replicated data management**. The ability to decide which copy of a replicated data item to access and to maintain the consistency of copies of a replicated data item.

- **Distributed database recovery**. The ability to recover from individual site crashes and from new types of failures, such as the failure of communication links.

- **Security**. Distributed transactions must be executed with the proper management of the security of the data and the authorization/access privileges of users.

- **Distributed directory (catalog) management**. A directory contains information (metadata) about data in the database. The directory may be global for the entire DDB, or local for each site. The placement and distribution of the directory are design and policy issues.

These functions themselves increase the complexity of a DDBMS over a centralized DBMS. Before we can realize the full potential advantages of distribution, we must find satisfactory solutions to these design issues and problems. Including all this additional functionality is hard to accomplish, and finding optimal solutions is a step beyond that.

**1.4. Baze de date NoSQL**

NoSQL encompasses a wide variety of different database technologies that were developed in response to a rise in the volume of data stored about users, objects and products, the frequency in which this data is accessed, and performance and processing needs. Relational databases, on the other hand, were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the cheap storage and processing power available today.

**1.4.1. Tipuri de baze de date NoSQL**

* **Document databases** pair each key with a complex data structure known as a document. Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.
* **Graph stores** are used to store information about networks, such as social connections. Graph stores include Neo4J and HyperGraphDB.
* **Key-value stores** are the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value. Examples of key-value stores are Riak and Voldemort. Some key-value stores, such as Redis, allow each value to have a type, such as "integer", which adds functionality.
* **Wide-column stores** such as Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows.

**1.4.2. Avantaje**

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address:

* Large volumes of structured, semi-structured, and unstructured data
* Agile sprints, quick iteration, and frequent code pushes
* Object-oriented programming that is easy to use and flexible
* Efficient, scale-out architecture instead of expensive, monolithic architecture

## Dynamic Schemas

Relational databases require that schemas be defined before you can add data. For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance.

This fits poorly with agile development approaches, because each time you complete new features, the schema of your database often needs to change. So if you decide, a few iterations into development, that you'd like to store customers' favorite items in addition to their addresses and phone numbers, you'll need to add that column to the database, and then migrate the entire database to the new schema.

If the database is large, this is a very slow process that involves significant downtime. If you are frequently changing the data your application stores – because you are iterating rapidly – this downtime may also be frequent. There's also no way, using a relational database, to effectively address data that's completely unstructured or unknown in advance.

NoSQL databases are built to allow the insertion of data without a predefined schema. That makes it easy to make significant application changes in real-time, without worrying about service interruptions – which means development is faster, code integration is more reliable, and less database administrator time is needed.

**1.4.3. NoSQL vs SQL**

|  |  |  |
| --- | --- | --- |
|  | SQL Databases | NOSQL Databases |
| Types | One type (SQL database) with minor variations | Many different types including key-value stores, document databases, wide-column stores, and graph databases |
| Development History | Developed in 1970s to deal with first wave of data storage applications | Developed in 2000s to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage |
| Examples | MySQL, Postgres, Oracle Database | MariaDB, MongoDB, Cassandra, HBase, Neo4j |
| Schemas | Structure and data types are fixed in advance. To store information about a new data item, the entire database must be altered, during which time the database must be taken offline | Typically dynamic. Records can add new information on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary. For some databases (e.g., wide-column stores), it is somewhat more challenging to add new fields dynamically. |
| Scaling | Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required. | Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary. |
| Development Model | Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database) | Open-source |
| Supports Transactions | Yes, updates can be configured to complete entirely or not at all | In certain circumstances and at certain levels (e.g., document level vs. database level) |
| Data Manipulation | Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE… | Through object-oriented APIs |