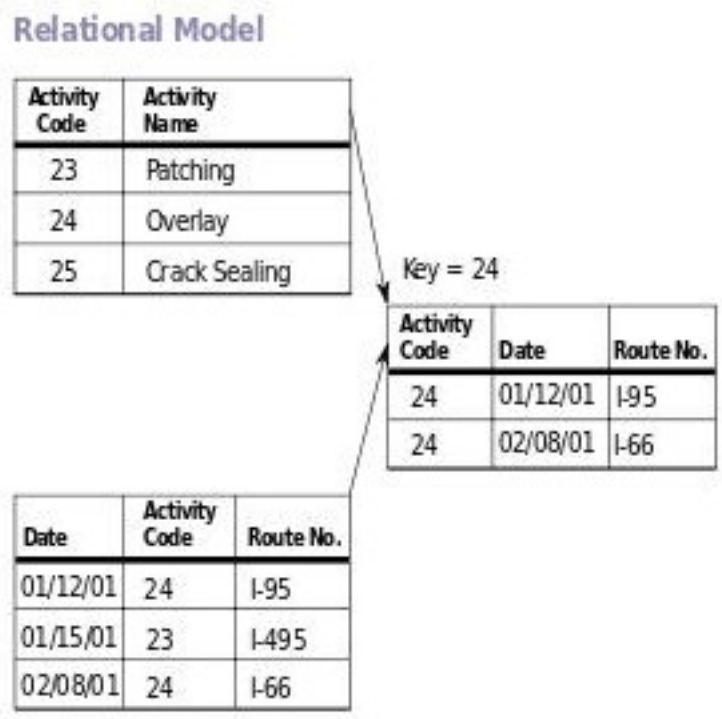
Unit 1: The Relational Model of Data and RDBMS Implementation Techniques

1.1 Theoretical concepts

The **relational model** for database management is a database model based on first-order predicate logic, first formulated and proposed in 1969 by Edgar F. Codd. In the relational model of a database, all data is represented in terms of tuples, grouped into relations. A database organized in terms of the relational model is a relational database.

The purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries.

Most relational databases use the SQL data definition and query language; these systems implement what can be regarded as an engineering approximation to the relational model. A *table* in an SQL database schema corresponds to a predicate variable; the contents of a table to a relation; key constraints, other constraints, and SQL queries correspond to predicates. However, SQL databases, including DB2, deviate from the relational model in many details, and Codd fiercely argued against deviations that compromise the original principles.



Alternatives to the relational model

Other models are the hierarchical model and network model. Some systems using these older architectures are still in use today in data centers with high data volume needs, or where existing systems are so complex and abstract it would be cost-prohibitive to migrate to systems employing the relational model; also of note are newer object-oriented databases.

Implementation

There have been several attempts to produce a true implementation of the relational database model as originally defined by Codd and explained by Date, Darwen and others, but none have been popular successes so far. Rel is one of the more recent attempts to do this.

The relational model was the first database model to be described in formal mathematical terms. Hierarchical and network databases existed before relational databases, but their specifications were relatively informal. After the relational model was defined, there were many attempts to compare and contrast the different models, and this led to the emergence of more rigorous descriptions of the earlier models; though the procedural nature of the data manipulation interfaces for hierarchical and network databases limited the scope for formalization.

A **relational database management system** (**RDBMS**) is a database management system (DBMS) that is based on the relational model as introduced by E. F. Codd, of IBM's San Jose Research Laboratory. Many popular databases currently in use are based on the relational database model.

RDBMSs have become[*when?*] a predominant choice for the storage of information in new databases used for financial records, manufacturing and logistical information, personnel data, and much more. Relational databases have often replaced legacy hierarchical databases and network databases because they are easier to understand and use. However, relational databases have been challenged by object databases, which were introduced in an attempt to address the object-relational impedance mismatch in relational database, and XML databases.

Database normalization

Relations are classified based upon the types of anomalies to which they're vulnerable. A database that's in

the first normal form is vulnerable to all types of anomalies, while a database that's in the domain/key normal form has no modification anomalies. Normal forms are hierarchical in nature. That is, the lowest level is the first normal form, and the database cannot meet the requirements for higher level normal forms without first having met all the requirements of the lesser normal forms.

Examples
An idealized, very simple example of a description of some relvars (relation variables) and their attributes:

- Customer (**Customer ID**, Tax ID, Name, Address, City, State, Zip, Phone, Email)
- Order (**Order No**, Customer ID, Invoice No, Date Placed, Date Promised, Terms, Status)
- Order Line (**Order No**, **Order Line No**, Product Code, Qty)
- Invoice (**Invoice No**, Customer ID, Order No, Date, Status)
- Invoice Line (**Invoice No**, **Invoice Line No**, Product Code, Qty Shipped)
- Product (**Product Code**, Product Description)

In this design we have six relvars: Customer, Order, Order Line, Invoice, Invoice Line and Product. The bold, underlined attributes are *candidate keys*. The non-bold, underlined attributes are *foreign keys*.

Usually one candidate key is arbitrarily chosen to be called the primary key and used in preference over the other candidate keys, which are then called alternate keys.

A *candidate key* is a unique identifier enforcing that no tuple will be duplicated; this would make the relation into something else, namely a bag, by violating the basic definition of a set. Both foreign keys and superkeys (that includes candidate keys) can be composite, that is, can be composed of several attributes. Below is a tabular depiction of a relation of our example Customer relvar; a relation can be thought of as a value that can be attributed to a relvar.

Customer relation

Customer ID	Tax ID	Name	Address	[More fields…]
1234567890	**555-5512222**	**Munmun**	**323 Broadway**	**…**

2223344556	**555-5523232**	**Wile E.**	**1200 Main Street**	**…**
3334445563	**555-5533323**	**Ekta**	**871 1st Street**	**…**
4232342432	**555-5325523**	**E. F. Codd**	**123 It Way**	**…**

If we attempted to *insert* a new customer with the ID *1234567890*, this would violate the design of the relvar since **Customer ID** is a *primary key* and we already have a customer *1234567890*. The DBMS must reject a transaction such as this that would render the database inconsistent by a violation of an integrity constraint.

Foreign keys are integrity constraints enforcing that the value of the attribute set is drawn from a *candidate key* in another relation. For example in the Order relation the attribute **Customer ID** is a foreign key. A *join* is the operation that draws on information from several relations at once. By joining relvars from the example above we could*query* the database for all of the Customers, Orders, and Invoices. If we only wanted the tuples for a specific customer, we would specify this using a restriction condition.

If we wanted to retrieve all of the Orders for Customer *1234567890*, we could query the database to return every row in the Order table with **Customer ID** *1234567890* and join the Order table to the Order Line table based on **Order No**.

There is a flaw in our database design above. The Invoice relvar contains an Order No attribute. So, each tuple in the Invoice relvar will have one Order No, which implies that there is precisely one Order for each Invoice. But in reality an invoice can be created against many orders, or indeed for no particular order. Additionally the Order relvar contains an Invoice No attribute, implying that each Order has a corresponding Invoice. But again this is not always true in the real world. An order is sometimes paid through several invoices, and sometimes paid without an invoice. In other words there can be many Invoices per Order and many Orders per Invoice. This is a **many-to-many** relationship between Order and Invoice (also called a *non-specific relationship*). To represent this relationship in the database a new relvar should be introduced whose role is to specify the correspondence between Orders and Invoices:

OrderInvoice(**Order No**,**Invoice No**)

Now, the Order relvar has a *one-to-many relationship* to the OrderInvoice table, as does the Invoice relvar. If we want to retrieve every Invoice for a particular Order, we can query for all orders where **Order No** in the Order relation equals the **Order No** in OrderInvoice, and where **Invoice No** in OrderInvoice equals the **Invoice No** in Invoice.

1.2 Relational model conformity and Integrity

Integrity constraints are used to ensure accuracy and consistency of data in a relational database. Data integrity is handled in a relational database through the concept of referential integrity. There are many types of integrity constraints that play a role in referential integrity.

Entity Integrity

The entity integrity constraint states that no primary key value can be null. This is because the primary key value is used to identify individual tuples in a relation. Having null value for the primary key implies that we cannot identify some tuples.This also specifies that there may not be any duplicate entries in primary key column key word.

Referential Integrity

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples in the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. It is a rule that maintains consistency among the rows of the two relations.

Domain Integrity

The domain integrity states that every element from a relation should respect the type and restrictions of its

corresponding attribute. A type can have a variable length which needs to be respected. Restrictions could be the range of values that the element can have, the default value if none is provided, and if the element can be NULL.

User Defined Integrity
A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business. E.g.: Age>=18 && Age<=60

1.3 Advanced SQL programming

SQL Structured Query Language is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS).

Originally based upon relational algebra and tuple relational calculus, SQL consists of a data definition language and a data manipulation language. The scope of SQL includes data insert, query, update and delete, schema creation and modification, and data access control. Although SQL is often described as, and to a great extent is, a declarative language (4GL), it also includes procedural elements.

SQL was one of the first commercial languages for Edgar F. Codd's relational model, as described in his influential 1970 paper "A Relational Model of Data for Large Shared Data Banks". Despite not entirely adhering to the relational model as described by Codd, it became the most widely used database language.

SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standards (ISO) in 1987. Since then, the standard has been enhanced several times with added features. But code is not completely portable among different database systems, which can lead to vendor lock-in. The different makers do not perfectly follow the standard, they add extensions, and the standard is sometimes ambiguous.

The SQL language is subdivided into several language elements, including:

- *Clauses*, which are constituent components of statements and queries. (In some cases, these are optional.)
- *Expressions*, which can produce either scalar values, or tables consisting of columns androws of data.
- *Predicates*, which specify conditions that can be evaluated to SQL three-valued logic (3VL)

(true/false/unknown) or Boolean truth values and which are used to limit the effects of statements and queries, or to change program flow.

•*Queries*, which retrieve the data based on specific criteria. This is an important element of*SQL*.

•*Statements*, which may have a persistent effect on schemata and data, or which may control transactions, program flow, connections, sessions, or diagnostics.

•SQL statements also include the semicolon (";") statement terminator. Though not required on every platform, it is defined as a standard part of the SQL grammar.

•*Insignificant whitespace* is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

Operators

Operator	Description	Example
=	**Equal to**	**`Author = 'Alcott'`**
<> *or* !=	**Not equal to**	**`Dept <> 'Sales'`**
>	**Greater than**	**`Hire_Date > '2012-01-31'`**
<	**Less than**	**`Bonus < 50000.00`**
>=	**Greater than or equal**	**`Dependents >= 2`**
<=	**Less than or equal**	**`Rate <= 0.05`**

`BETWEEN`	Between an inclusive range	`Cost BETWEEN 100.00 AND 500.00`
`LIKE`	Match a character pattern	`First_Name LIKE 'Will%'`
`IN`	Equal to one of multiple possible values	`DeptCode IN (101, 103, 209)`
`IS` *or* `IS NOT`	Compare to null (missing data)	`Address IS NOT NULL`

Conditional (CASE) expressions

SQL has a case/when/then/else/end expression, which was introduced in SQL-92. In its most general form, which is called a "searched case" in the SQL standard, it works like else if in other programming languages:

```
CASE WHEN n > 0
        THEN 'positive'
     WHEN n < 0
        THEN 'negative'
     ELSE 'zero'
END
```

The WHEN conditions are tested in the order in which they appear in the source. If no ELSE expression is specified, it defaults to ELSE NULL. An abbreviated syntax exists mirroring switch statements; it is called "simple case" in the SQL standard:

```
CASE n WHEN 1
          THEN 'one'
       WHEN 2
          THEN 'two'
       ELSE 'i cannot count that high'
```

```
END
```
This syntax uses implicit equality comparisons, with the usual caveats for comparing with NULL.

For the Oracle-SQL dialect, the latter can be shortened to an equivalent DECODE construct:

```
SELECT DECODE(n, 1, "one",
              2, "two",
              "i cannot count that high")
FROM   some_table;
```
The last value is the default; if none is specified, it also defaults to NULL. However, unlike the standard's "simple case", Oracle's DECODE considers two NULLs to be equal with each other.

Queries

The most common operation in SQL is the query, which is performed with the declarative SELECT statement. SELECT retrieves data from one or more tables, or expressions. Standard SELECT statements have no persistent effects on the database. Some non-standard implementations of SELECT can have persistent effects, such as the SELECT INTO syntax that exists in some databases.

Queries allow the user to describe desired data, leaving the database management system (DBMS) responsible for planning, optimizing, and performing the physical operations necessary to produce that result as it chooses.

A query includes a list of columns to be included in the final result immediately following the SELECT keyword. An asterisk ("*") can also be used to specify that the query should return all columns of the queried tables. SELECT is the most complex statement in SQL, with optional keywords and clauses that include:

- The FROM clause which indicates the table(s) from which data is to be retrieved. The FROM clause can include optional JOIN subclauses to specify the rules for joining tables.
- The WHERE clause includes a comparison predicate, which restricts the rows returned by the query. The WHERE clause eliminates all rows from the result set for which the comparison predicate does not evaluate to True.
- The GROUP BY clause is used to project rows having common values into a smaller set of

rows. GROUP BY is often used in conjunction with SQL aggregation functions or to eliminate duplicate rows from a result set. The WHERE clause is applied before the GROUP BY clause.

•The HAVING clause includes a predicate used to filter rows resulting from the GROUP BY clause. Because it acts on the results of the GROUP BY clause, aggregation functions can be used in the HAVING clause predicate.

•The ORDER BY clause identifies which columns are used to sort the resulting data, and in which direction they should be sorted (options are ascending or descending). Without an ORDER BY clause, the order of rows returned by an SQL query is undefined.

The following is an example of a SELECT query that returns a list of expensive books. The query retrieves all rows from the *Book* table in which the *price* column contains a value greater than 100.00. The result is sorted in ascending order by *title*. The asterisk (*) in the *select list* indicates that all columns of the *Book* table should be included in the result set.

```
SELECT *
 FROM  Book
 WHERE price > 100.00
 ORDER BY title;
```

The example below demonstrates a query of multiple tables, grouping, and aggregation, by returning a list of books and the number of authors associated with each book.

```
SELECT Book.title AS Title,
     COUNT(*) AS Authors
 FROM  Book
 JOIN  Book_author
   ON  Book.isbn = Book_author.isbn
 GROUP BY Book.title;
```

Example output might resemble the following:

```
Title                Authors

-------------------- -------
```

```
SQL Examples and Guide 4

The Joy of SQL         1

An Introduction to SQL 2

Pitfalls of SQL        1
```

Under the precondition that *isbn* is the only common column name of the two tables and that a column named *title* only exists in the *Books* table, the above query could be rewritten in the following form:

```
SELECT title,
    COUNT(*) AS Authors
FROM  Book
NATURAL JOIN Book_author
GROUP BY title;
```

However, many vendors either do not support this approach, or require certain column naming conventions in order for natural joins to work effectively.

SQL includes operators and functions for calculating values on stored values. SQL allows the use of expressions in the *select list* to project data, as in the following example which returns a list of books that cost more than 100.00 with an additional *sales_tax* column containing a sales tax figure calculated at 6% of the *price*.

```
SELECT isbn,
    title,
    price,
    price * 0.06 AS sales_tax
FROM  Book
WHERE price > 100.00
ORDER BY title;
```

Subqueries

Queries can be nested so that the results of one query can be used in another query via a relational operator or aggregation function. A nested query is also known as a *subquery*. While joins and other table operations provide computationally superior (i.e. faster) alternatives in many cases, the use of subqueries introduces a hierarchy in execution which can be useful or necessary. In the following example, the aggregation function AVG receives as input the result of a subquery:

```
SELECT isbn,title,price
 FROM  Book
 WHERE price < (SELECT AVG(price) FROM Book)
 ORDER BY title;
```

A subquery can use values from the outer query, in which case it is known as a correlated subquery.

Since 1999 the SQL standard allows named subqueries called common table expression (named and designed after the IBM DB2 version 2 implementation; Oracle calls these subquery factoring). CTEs can be also be recursive by referring to themselves; the resulting mechanism allows tree or graph traversals (when represented as relations), and more generally fixpoint computations.

Null and three-valued logic (3VL)

The concept of Null was introduced into SQL to handle missing information in the relational model. The word NULL is a reserved keyword in SQL, used to identify the Null special marker. Comparisons with Null, for instance equality (=) in WHERE clauses, results in an Unknown truth value. In SELECT statements SQL returns only results for which the WHERE clause returns a value of True; i.e. it excludes results with values of False and also excludes those whose value is Unknown.

Along with True and False, the Unknown resulting from direct comparisons with Null thus brings a fragment of three-valued logic to SQL. The truth tables SQL uses for AND, OR, and NOT correspond to a common fragment of the Kleene and Lukasiewicz three-valued logic (which differ in their definition of implication, however SQL defines no such operation).

There are however disputes about the semantic interpretation of Nulls in SQL because of its treatment outside direct comparisons. As seen in the table above direct equality comparisons between two NULLs in SQL (e.g. NULL = NULL) returns a truth value of Unknown. This is in line with the interpretation that Null does not have a value (and is not a member of any data domain) but is rather a placeholder or "mark" for

missing information. However, the principle that two Nulls aren't equal to each other is effectively violated in the SQL specification for the UNION and INTERSECT operators, which do identify nulls with each other. Consequently, these set operations in SQL may produce results not representing sure information, unlike operations involving explicit comparisons with NULL (e.g. those in a WHERE clause discussed above). In Codd's 1979 proposal (which was basically adopted by SQL92) this semantic inconsistency is rationalized by arguing that removal of duplicates in set operations happens "at a lower level of detail than equality testing in the evaluation of retrieval operations." However, computer science professor Ron van der Meyden concluded that "The inconsistencies in the SQL standard mean that it is not possible to ascribe any intuitive logical semantics to the treatment of nulls in SQL."

Additionally, since SQL operators return Unknown when comparing anything with Null directly, SQL provides two Null-specific comparison predicates: IS NULL and IS NOT NULL test whether data is or is not Null. Universal quantification is not explicitly supported by SQL, and must be worked out as a negated existential quantification. There is also the "<row value expression> IS DISTINCT FROM <row value expression>" infixed comparison operator which returns TRUE unless both operands are equal or both are NULL. Likewise, IS NOT DISTINCT FROM is defined as "NOT (<row value expression> IS DISTINCT FROM <row value expression>)". SQL:1999 also introduced BOOLEAN type variables, which according to the standard can also hold Unknown values. In practice, a number of systems (e.g. PostgreSQL) implement the BOOLEAN Unknown as a BOOLEAN NULL.

Data manipulation

The Data Manipulation Language (DML) is the subset of SQL used to add, update and delete data:

- INSERT adds rows (formally tuples) to an existing table, e.g.:

```
INSERT INTO example
 (field1, field2, field3)
 VALUES
 ('test', 'N', NULL);
```

- UPDATE modifies a set of existing table rows, e.g.:

```
UPDATE example
 SET field1 = 'updated value'
 WHERE field2 = 'N';
```

•DELETE removes existing rows from a table, e.g.:

```
DELETE FROM example
 WHERE field2 = 'N';
```

•MERGE is used to combine the data of multiple tables. It combines the INSERT and UPDATE elements. It is defined in the SQL:2003 standard; prior to that, some databases provided similar functionality via different syntax, sometimes called "upsert".

```
 MERGE INTO TABLE_NAME USING table_reference ON (condition)
 WHEN MATCHED THEN
 UPDATE SET column1 = value1 [, column2 = value2 ...]
 WHEN NOT MATCHED THEN
 INSERT (column1 [, column2 ...]) VALUES (value1 [, value2 ...
```

Transaction controls

Transactions, if available, wrap DML operations:

•START TRANSACTION (or BEGIN WORK, or BEGIN TRANSACTION, depending on SQL dialect) marks the start of a database transaction, which either completes entirely or not at all.

•SAVE TRANSACTION (or SAVEPOINT) saves the state of the database at the current point in transaction

```
CREATE TABLE tbl_1(id INT);
 INSERT INTO tbl_1(id) VALUES(1);
 INSERT INTO tbl_1(id) VALUES(2);
COMMIT;
 UPDATE tbl_1 SET id=200 WHERE id=1;
SAVEPOINT id_1upd;
 UPDATE tbl_1 SET id=1000 WHERE id=2;
ROLLBACK TO id_1upd;
 SELECT id FROM tbl_1;
```

•COMMIT causes all data changes in a transaction to be made permanent.

•ROLLBACK causes all data changes since the last COMMIT or ROLLBACK to be discarded, leaving the state of the data as it was prior to those changes.

Once the COMMIT statement completes, the transaction's changes cannot be rolled back.

COMMIT and ROLLBACK terminate the current transaction and release data locks. In the absence of a START TRANSACTION or similar statement, the semantics of SQL are implementation-dependent. The following example shows a classic transfer of funds transaction, where money is removed from one account and added to another. If either the removal or the addition fails, the entire transaction is rolled back.

```
START TRANSACTION;
 UPDATE Account SET amount=amount-200 WHERE account_number=1234;
 UPDATE Account SET amount=amount+200 WHERE account_number=2345;

IF ERRORS=0 COMMIT;
IF ERRORS<>0 ROLLBACK;
```

Data definition

The Data Definition Language (DDL) manages table and index structure. The most basic items of DDL are the CREATE, ALTER, RENAME, DROP and TRUNCATEstatements:

- CREATE creates an object (a table, for example) in the database, e.g.:

```
CREATE TABLE example(
 field1 INTEGER,
 field2 VARCHAR(50),
 field3 DATE NOT NULL,
 PRIMARY KEY (field1, field2)
);
```

- ALTER modifies the structure of an existing object in various ways, for example, adding a column to an existing table or a constraint, e.g.:

```
ALTER TABLE example ADD field4 NUMBER(3) NOT NULL;
```

- TRUNCATE deletes all data from a table in a very fast way, deleting the data inside the table and not the table itself. It usually implies a subsequent COMMIT operation, i.e., it cannot be rolled back (data is not written to the logs for rollback later, unlike DELETE).

```
TRUNCATE TABLE example;
```

- DROP deletes an object in the database, usually irretrievably, i.e., it cannot be rolled back, e.g.:

DROP TABLE example;

Data types

Each column in an SQL table declares the type(s) that column may contain. ANSI SQL includes the following data types.

Character strings

- CHARACTER(n) or CHAR(n): fixed-width n-character string, padded with spaces as needed
- CHARACTER VARYING(n) or VARCHAR(n): variable-width string with a maximum size of n characters
- NATIONAL CHARACTER(n) or NCHAR(n): fixed width string supporting an international character set
- NATIONAL CHARACTER VARYING(n) or NVARCHAR(n): variable-width NCHAR string

Bit strings

- BIT(n): an array of n bits
- BIT VARYING(n): an array of up to n bits

Numbers

- INTEGER and SMALLINT
- FLOAT, REAL and DOUBLE PRECISION
- NUMERIC(precision, scale) or DECIMAL(precision, scale)

For example, the number 123.45 has a precision of 5 and a scale of 2. The precision is a positive integer that determines the number of significant digits in a particular radix (binary or decimal). The scale is a non-negative integer. A scale of 0 indicates that the number is an integer. For a decimal number with scale S, the exact numeric value is the integer value of the significant digits divided by 10S.

SQL provides a function to round numerics or dates, called TRUNC (in Informix, DB2, PostgreSQL, Oracle and MySQL) or ROUND (in Informix, SQLite, Sybase, Oracle, PostgreSQL and Microsoft SQL Server).

Date and time

- DATE: for date values (e.g. 2011-05-03)

- TIME: for time values (e.g. 15:51:36). The granularity of the time value is usually a *tick* (100 nanoseconds).
- TIME WITH TIME ZONE or TIMETZ: the same as TIME, but including details about the time zone in question.
- TIMESTAMP: This is a DATE and a TIME put together in one variable (e.g. 2011-05-03 15:51:36).
- TIMESTAMP WITH TIME ZONE or TIMESTAMPTZ: the same as TIMESTAMP, but including details about the time zone in question.

SQL provides several functions for generating a date / time variable out of a date / time string (TO_DATE, TO_TIME, TO_TIMESTAMP), as well as for extracting the respective members (seconds, for instance) of such variables. The current system date / time of the database server can be called by using functions like NOW.

Data control

The Data Control Language (DCL) authorizes users to access and manipulate data. Its two main statements are:

- GRANT authorizes one or more users to perform an operation or a set of operations on an object.
- REVOKE eliminates a grant, which may be the default grant.

Example:

```
GRANT SELECT, UPDATE
 ON example
 TO some_user, another_user;

REVOKE SELECT, UPDATE
 ON example
 FROM some_user, another_user;
```

1.4 Query optimization

Query optimization is a function of many relational database management systems. The **query optimizer** attempts to determine the most efficient way to execute a given query by considering the possible query plans.

Generally, the query optimizer cannot be accessed directly by users: once queries are submitted to database server, and parsed by the parser, they are then passed to the query optimizer where optimization occurs. However, some database engines allow guiding the query optimizer with hints.

A query is a request for information from a database. It can be as simple as "finding the address of a person with SS# 123-45-6789," or more complex like "finding the average salary of all the employed married men in California between the ages 30 to 39, that earn less than their wives." Queries results are generated by accessing relevant database data and manipulating it in a way that yields the requested information. Since database structures are complex, in most cases, and especially for not-very-simple queries, the needed data for a query can be collected from a database by accessing it in different ways, through different data-structures, and in different orders. Each different way typically requires different processing time. Processing times of a same query may have large variance, from a fraction of a second to hours, depending on the way selected. The purpose of query optimization, which is an automated process, is to find the way to process a given query in minimum time. The large possible variance in time justifies performing query optimization, though finding the exact optimal way to execute a query, among all possibilities, is typically very complex, time consuming by itself, may be too costly, and often practically impossible. Thus query optimization typically tries to approximate the optimum by comparing several common-sense alternatives to provide in a reasonable time a "good enough" plan which typically does not deviate much from the best possible result.

1.5 Concurrency control and Transaction management

In information technology and computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

Computer systems, both software and hardware, consist of modules, or components. Each component is

designed to operate correctly, i.e., to obey or to meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction. Operation consistency and correctness should be achieved with as good as possible efficiency, without reducing performance below reasonable.

For example, a failure in concurrency control can result in data corruption from torn read or write operations.

Concurrency control in Database management systems (DBMS; e.g., Bernstein et al. 1987, Weikum and Vossen 2001), other transactional objects, and related distributed applications (e.g., Grid computing and Cloud computing) ensures that *database transactions* are performed concurrently without violating the data integrity of the respective databases. Thus concurrency control is an essential element for correctness in any system where two database transactions or more, executed with time overlap, can access the same data, e.g., virtually in any general-purpose database system. Consequently a vast body of related research has been accumulated since database systems emerged in the early 1970s. A well established concurrency control theory for database systems is outlined in the references mentioned above:serializability theory, which allows to effectively design and analyze concurrency control methods and mechanisms. An alternative theory for concurrency control of atomic transactions over abstract data types is presented in (Lynch et al. 1993), and not utilized below. This theory is more refined, complex, with a wider scope, and has been less utilized in the Database literature than the classical theory above. Each theory has its pros and cons, emphasis and insight. To some extent they are complementary, and their merging may be useful.

To ensure correctness, a DBMS usually guarantees that only *serializable* transaction schedules are generated, unless *serializability* is intentionally relaxed to increase performance, but only in cases where application correctness is not harmed. For maintaining correctness in cases of failed (aborted) transactions (which can always happen for many reasons) schedules also need to have the *recoverability* (from abort) property. A DBMS also guarantees that no effect of *committed* transactions is lost, and no effect of *aborted* (rolled back) transactions remains in the related database. Overall transaction characterization is

usually summarized by the ACID rules below. As databases have become distributed, or needed to cooperate in distributed environments (e.g., Federated databases in the early 1990, and Cloud computingcurrently), the effective distribution of concurrency control mechanisms has received special attention.

Database transaction and the ACID rules

The concept of a *database transaction* (or *atomic transaction*) has evolved in order to enable both a well understood database system behavior in a faulty environment where crashes can happen any time, and *recovery* from a crash to a well understood database state. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands). Every database transaction obeys the following rules (by support in the database system; i.e., a database system is designed to guarantee them for the transactions it runs):

•**Atomicity** - Either the effects of all or none of its operations remain ("all or nothing" semantics) when a transaction is completed (*committed* or *aborted* respectively). In other words, to the outside world a committed transaction appears (by its effects on the database) to be indivisible, atomic, and an aborted transaction does not leave effects on the database at all, as if never existed.

•**Consistency** - Every transaction must leave the database in a consistent (correct) state, i.e., maintain the predetermined integrity rules of the database (constraints upon and among the database's objects). A transaction must transform a database from one consistent state to another consistent state (however, it is the responsibility of the transaction's programmer to make sure that the transaction itself is correct, i.e., performs correctly what it intends to perform (from the application's point of view) while the predefined integrity rules are enforced by the DBMS). Thus since a database can be normally changed only by transactions, all the database's states are consistent. An aborted transaction does not change the database state it has started from, as if it never existed (atomicity above).

- **Isolation** - Transactions cannot interfere with each other (as an end result of their executions). Moreover, usually (depending on concurrency control method) the effects of an incomplete transaction are not even visible to another transaction. Providing isolation is the main goal of

concurrency control.

- **Durability** - Effects of successful (committed) transactions must persist through crashes (typically by recording the transaction's effects and its commit event in a non-volatile memory).

The concept of atomic transaction has been extended during the years to what has become Business transactions which actually implement types of Workflow and are not atomic. However also such enhanced transactions typically utilize atomic transactions as components.

Why is concurrency control needed?

If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur. Here are some typical examples:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistent.

Concurrency control mechanisms

Categories

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations ("...and be optimistic about the rules being met..."), and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead (versus executing it to the end only once). If not too many transactions are aborted, then being optimistic is usually a good strategy.
- **Pessimistic** - Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.
- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

Different categories provide different performance, i.e., different average transaction completion rates (*throughput*), depending on transaction types mix, computing level of parallelism, and other factors. If selection and knowledge about trade-offs are available, then category and method should be chosen to provide the highest performance.

The mutual blocking between two transactions (where each one blocks the other) or more results in a deadlock, where the transactions involved are stalled and cannot reach completion. Most non-optimistic mechanisms (with blocking) are prone to deadlocks which are resolved by an intentional abort of a stalled transaction (which releases the other transactions in that deadlock), and its immediate restart and re-execution. The likelihood of a deadlock is typically low.

Both blocking, deadlocks, and aborts result in performance reduction, and hence the trade-offs between the categories.

Methods

Many methods for concurrency control exist. Most of them can be implemented within either main category above. The major methods, which have each many variants, and in some cases may overlap or be combined, are:

1.Locking (e.g., **Two-phase locking** - 2PL) - Controlling access to data by locks assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release.

2.Serialization graph checking (also called Serializability, or Conflict, or Precedence graph checking) - Checking for cycles in the schedule's graph and breaking them by aborts.

3.Timestamp ordering (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.

4.Commitment ordering (or Commit ordering; CO) - Controlling or checking transactions' chronological order of commit events to be compatible with their respective precedence order.

Other major concurrency control types that are utilized in conjunction with the methods above include:

•**Multiversion concurrency control** (MVCC) - Increasing concurrency and performance by generating a new version of a database object each time the object is written, and allowing transactions' read operations of several last relevant versions (of each object) depending on scheduling method.

•**Index concurrency control** - Synchronizing access operations to indexes, rather than to user data. Specialized methods provide substantial performance gains.

•**Private workspace model** (**Deferred update**) - Each transaction maintains a private workspace for its accessed data, and its changed data become visible outside the transaction only upon its commit (e.g., Weikum and Vossen 2001). This model provides a different concurrency control behavior with benefits in many cases.

The most common mechanism type in database systems since their early days in the 1970s has been *Strong strict Two-phase locking* (SS2PL; also called *Rigorous scheduling* or *Rigorous 2PL*) which is a special case (variant) of both Two-phase locking (2PL) and Commitment ordering (CO). It is pessimistic. In spite of its long name (for historical reasons) the idea of the **SS2PL** mechanism is simple: "Release all locks applied by a transaction only after the transaction has ended." SS2PL (or Rigorousness) is also the name of the set of all schedules that can be generated by this mechanism, i.e., these are SS2PL (or Rigorous) schedules, have the SS2PL (or Rigorousness) property.

Major goals of concurrency control mechanisms

Concurrency control mechanisms firstly need to operate correctly, i.e., to maintain each transaction's integrity rules (as related to concurrency; application-specific integrity rule are out of the scope here) while

transactions are running concurrently, and thus the integrity of the entire transactional system. Correctness needs to be achieved with as good performance as possible. In addition, increasingly a need exists to operate effectively while transactions are distributed over processes,computers, and computer networks. Other subjects that may affect concurrency control are recovery and replication.

Correctness

Serializability

For correctness, a common major goal of most concurrency control mechanisms is generating schedules with the *Serializability* property. Without serializability undesirable phenomena may occur, e.g., money may disappear from accounts, or be generated from nowhere. **Serializability** of a schedule means equivalence (in the resulting database values) to some *serial* schedule with the same transactions (i.e., in which transactions are sequential with no overlap in time, and thus completely isolated from each other: No concurrent access by any two transactions to the same data is possible). Serializability is considered the highest level of isolation amongdatabase transactions, and the major correctness criterion for concurrent transactions. In some cases compromised, relaxed forms of serializability are allowed for better performance (e.g., the popular *Snapshot isolation* mechanism) or to meet availability requirements in highly distributed systems (see *Eventual consistency*), but only if application's correctness is not violated by the relaxation (e.g., no relaxation is allowed for money transactions, since by relaxation money can disappear, or appear from nowhere).

Almost all implemented concurrency control mechanisms achieve serializability by providing *Conflict serializablity*, a broad special case of serializability (i.e., it covers, enables most serializable schedules, and does not impose significant additional delay-causing constraints) which can be implemented efficiently.

Recoverability

Concurrency control typically also ensures the *Recoverability* property of schedules for maintaining correctness in cases of aborted transactions (which can always happen for many reasons). **Recoverability** (from abort) means that no committed transaction in a schedule has read data written by an aborted transaction. Such data disappear from the database (upon the abort) and are parts of an incorrect database state. Reading such data violates the consistency rule of ACID. Unlike Serializability, Recoverability cannot be compromised, relaxed at any case, since any relaxation results in quick database integrity violation upon aborts. The major methods listed above provide serializability mechanisms. None of

them in its general form automatically provides recoverability, and special considerations and mechanism enhancements are needed to support recoverability. A commonly utilized special case of recoverability is *Strictness*, which allows efficient database recovery from failure (but excludes optimistic implementations; e.g., Strict CO (SCO) cannot have an optimistic implementation, but has semi-optimistic ones).

Comment: Note that the *Recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is rather needed to correctly automatically handle transaction aborts, which may be unrelated to database failure and recovery from it.

Distribution

With the fast technological development of computing the difference between local and distributed computing over low latency networks or buses is blurring. Thus the quite effective utilization of local techniques in such distributed environments is common, e.g., in computer clusters and multi-core processors. However the local techniques have their limitations and use multi-processes (or threads) supported by multi-processors (or multi-cores) to scale. This often turns transactions into distributed ones, if they themselves need to span multi-processes. In these cases most local concurrency control techniques do not scale well.

Distributed serializability and Commitment ordering

As database systems have become distributed, or started to cooperate in distributed environments (e.g., Federated databases in the early 1990s, and nowadays Grid computing, Cloud computing, and networks with smartphones), some transactions have become distributed. A distributed transaction means that the transaction spansprocesses, and may span computers and geographical sites. This generates a need in effective distributed concurrency control mechanisms. Achieving the Serializability property of a distributed system's schedule (see *Distributed serializability* and *Global serializability* (*Modular serializability*)) effectively poses special challenges typically not met by most of the regular serializability mechanisms, originally designed to operate locally. This is especially due to a need in costly distribution of concurrency control information amid communication and computer latency. The only known general effective technique for distribution is Commitment ordering, which was disclosed publicly in 1991 (after being patented). **Commitment ordering** (Commit ordering, CO; Raz 1992) means that transactions' chronological order of commit events is kept compatible with their respective precedence order. CO does not require the distribution of concurrency control information and provides a general effective solution

(reliable, high-performance, and scalable) for both distributed and global serializability, also in a heterogeneous environment with database systems (or other transactional objects) with different (any) concurrency control mechanisms.[1] CO is indifferent to which mechanism is utilized, since it does not interfere with any transaction operation scheduling (which most mechanisms control), and only determines the order of commit events. Thus, CO enables the efficient distribution of all other mechanisms, and also the distribution of a mix of different (any) local mechanisms, for achieving distributed and global serializability. The existence of such a solution has been considered "unlikely" until 1991, and by many experts also later, due to misunderstanding of the CO solution (see Quotations in *Global serializability*). An important side-benefit of CO is automatic distributed deadlock resolution. Contrary to CO, virtually all other techniques (when not combined with CO) are prone to distributed deadlocks (also called global deadlocks) which need special handling. CO is also the name of the resulting schedule property: A schedule has the CO property if the chronological order of its transactions' commit events is compatible with the respective transactions' precedence (partial) order.

SS2PL mentioned above is a variant (special case) of CO and thus also effective to achieve distributed and global serializability. It also provides automatic distributed deadlock resolution (a fact overlooked in the research literature even after CO's publication), as well as Strictness and thus Recoverability. Possessing these desired properties together with known efficient locking based implementations explains SS2PL's popularity. SS2PL has been utilized to efficiently achieve Distributed and Global serializability since the 1980, and has become the de facto standard for it. However, SS2PL is blocking and constraining (pessimistic), and with the proliferation of distribution and utilization of systems different from traditional database systems (e.g., as in Cloud computing), less constraining types of CO (e.g., Optimistic CO) may be needed for better performance.

Distributed recoverability

Unlike Serializability, *Distributed recoverability* and *Distributed strictness* can be achieved efficiently in a straightforward way, similarly to the way Distributed CO is achieved: In each database system they have to be applied locally, and employ a vote ordering strategy for the Two-phase commit protocol (2PC; Raz 1992, page 307).

As has been mentioned above, Distributed SS2PL, including Distributed strictness (recoverability) and Distributed commitment ordering (serializability), automatically employs the needed vote ordering strategy,

and is achieved (globally) when employed locally in each (local) database system (as has been known and utilized for many years; as a matter of fact locality is defined by the boundary of a 2PC participant (Raz 1992)).

Other major subjects of attention

The design of concurrency control mechanisms is often influenced by the following subjects:

Recovery

All systems are prone to failures, and handling *recovery* from failure is a must. The properties of the generated schedules, which are dictated by the concurrency control mechanism, may have an impact on the effectiveness and efficiency of recovery. For example, the Strictness property (mentioned in the section Recoverability above) is often desirable for an efficient recovery.

Replication

For high availability database objects are often *replicated*. Updates of replicas of a same database object need to be kept synchronized. This may affect the way concurrency control is done (e.g., Gray et al. 1996).

transaction processing

In computer science, **transaction processing** is information processing that is divided into individual, indivisible operations, called *transactions.* Each transaction must succeed or fail as a complete unit; it cannot remain in an intermediate state.

Since most, though not necessarily all, transaction processing today is interactive the term is often treated as synonymous with *online transaction processing*.

The basic principles of all transaction-processing systems are the same. However, the terminology may vary from one transaction-processing system to another, and the terms used below are not necessarily universal.

Rollback

Transaction-processing systems ensure database integrity by recording intermediate states of the database as it is modified, then using these records to restore the database to a known state if a transaction cannot be committed. For example, copies of information on the database *prior* to its modification by a transaction are set aside by the system before the transaction can make any modifications (this is sometimes called

a *before image*). If any part of the transaction fails before it is committed, these copies are used to restore the database to the state it was in before the transaction began.

Rollforward

It is also possible to keep a separate journal of all modifications to a database (sometimes called *after images*). This is not required for rollback of failed transactions but it is useful for updating the database in the event of a database failure, so some transaction-processing systems provide it. If the database fails entirely, it must be restored from the most recent back-up. The back-up will not reflect transactions committed since the back-up was made. However, once the database is restored, the journal of after images can be applied to the database (*rollforward*) to bring the database up to date. Any transactions in progress at the time of the failure can then be rolled back. The result is a database in a consistent, known state that includes the results of all transactions committed up to the moment of failure.

Deadlocks

In some cases, two transactions may, in the course of their processing, attempt to access the same portion of a database at the same time, in a way that prevents them from proceeding. For example, transaction A may access portion X of the database, and transaction B may access portion Y of the database. If, at that point, transaction A then tries to access portion Y of the database while transaction B tries to access portion X, a *deadlock* occurs, and neither transaction can move forward. Transaction-processing systems are designed to detect these deadlocks when they occur. Typically both transactions will be cancelled and rolled back, and then they will be started again in a different order, automatically, so that the deadlock doesn't occur again. Or sometimes, just one of the deadlocked transactions will be cancelled, rolled back, and automatically restarted after a short delay.

Deadlocks can also occur between three or more transactions. The more transactions involved, the more difficult they are to detect, to the point that transaction processing systems find there is a practical limit to the deadlocks they can detect.

Compensating transaction

In systems where commit and rollback mechanisms are not available or undesirable, a compensating transaction is often used to undo failed transactions and restore the system to a previous state.

ACID criteria

Jim Gray defined properties of a reliable transaction system in the late 1970s under the acronym *ACID* —

atomicity, consistency, isolation, and durability.

Atomicity
A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

Consistency
Consistency: A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation
Even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.

Durability
Once a transaction completes successfully (commits), its changes to the state survive failures.

Benefits
Transaction processing has these benefits:

- It allows sharing of computer resources among many users
- It shifts the time of job processing to when the computing resources are less busy
- It avoids idling the computing resources without minute-by-minute human interaction and supervision
- It is used on expensive classes of computers to help amortize the cost by keeping high rates of utilization of those expensive resources

1.6 Database performance tuning

Database tuning describes a group of activities used to optimize and homogenize the performance of a database. It usually overlaps with query tuning, but refers to design of the database files, selection of the database management system (DBMS), operating system and CPU the DBMS runs on.

Database tuning aims to maximize use of system resources to perform work as efficiently and rapidly as possible. Most systems are designed to manage their use of system resources, but there is still much room to improve their efficency by customizing their settings and configuration for the database and the DBMS.

I/O tuning

Hardware and software configuration of disk subsystems are examined: RAID levels and configuration, block and stripe size allocation, and the configuration of disks,controller cards, storage cabinets, and external storage systems such as SANs. Transaction logs and temporary spaces are heavy consumers of I/O, and affect performance for all users of the database. Placing them appropriately is crucial.

Frequently joined tables and indexes are placed so that as they are requested from file storage, they can be retrieved in parallel from separate disks simultaneously. Frequently accessed tables and indexes are placed on separate disks to balance I/O and prevent read queuing.

DBMS tuning

DBMS tuning refers to tuning of the DBMS and the configuration of the memory and processing resources of the computer running the DBMS. This is typically done through configuring the DBMS, but the resources involved are shared with the host system.

Tuning the DBMS can involve setting the recovery interval (time needed to restore the state of data to a particular point in time), assigning parallelism (the breaking up of work from a single query into tasks assigned to different processing resources), and network protocols used to communicate with database consumers.

Memory is allocated for data, execution plans, procedure cache, and work space[*clarify*]. It is much faster to access data in memory than data on storage, so maintaining a sizable cache of data makes activities perform faster. The same consideration is given to work space. Caching execution plans and procedures means that they are reused instead of recompiled when needed. It is important to take as much memory as possible, while leaving enough for other processes and the OS to use without excessive paging of memory to storage.

Processing resources are sometimes assigned to specific activities to improve concurrency. On a server with eight processors, six could be reserved for the DBMS to maximize available processing resources for the database.

Database maintenance

Database maintenance includes backups, column statistics updates, and defragmentation of data inside the database files.

On a heavily used database, the transaction log grows rapidly. Transaction log entries must be removed from the log to make room for future entries. Frequent transaction log backups are smaller, so they interrupt database activity for shorter periods of time.

DBMS use statistic histograms to find data in a range against a table or index. Statistics updates should be scheduled frequently and sample as much of the underlying data as possible. Accurate and updated statistics allow query engines to make good decisions about execution plans, as well as efficiently locate data.

Defragmentation of table and index data increases efficiency in accessing data. The amount of fragmentation depends on the nature of the data, how it is changed over time, and the amount of free space in database pages to accept inserts of data without creating additional pages.

1.7 Distributed relational systems and Data Replication

A **distributed database** is a database in which storage devices are not all attached to a common processing unit such as the CPU, controlled by a **distributeddatabase management system** (together sometimes called a **distributed database system**). It may be stored in multiple computers, located in the same physical location; or may be dispersed over a network of interconnected computers. Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system cdatabase, inheritanceonsists of loosely-coupled sites that share no physical components.

System administrators can distribute collections of data (e.g. in a database) across multiple physical locations. A distributed database can reside on network servers on the Internet, on corporate intranets or extranets, or on other company networks. Because they store data across multiple computers, distributed databases can improve performance at end-user worksites by allowing transactions to be processed on many machines, instead of being limited to one.

Two processes ensure that the distributed databases remain up-to-date and current: replication and duplication.

1. Replication involves using specialized software that looks for changes in the distributive database. Once the changes have been identified, the replication process makes all the databases look the same.

The replication process can be complex and time-consuming depending on the size and number of the distributed databases. This process can also require a lot of time and computer resources.

2.Duplication, on the other hand, has less complexity. It basically identifies one database as a master and then duplicates that database. The duplication process is normally done at a set time after hours. This is to ensure that each distributed location has the same data. In the duplication process, users may change only the master database. This ensures that local data will not be overwritten.

Both replication and duplication can keep the data current in all distributive locations.

Besides distributed database replication and fragmentation, there are many other distributed database design technologies. For example, local autonomy, synchronous and asynchronous distributed database technologies. These technologies' implementation can and does depend on the needs of the business and the sensitivity/confidentiality of the data stored in the database, and hence the price the business is willing to spend on ensuring data security, consistency and integrity.

When discussing access to distributed databases, Microsoft favors the term **distributed query**, which it defines in protocol-specific manner as "[a]ny SELECT, INSERT, UPDATE, or DELETE statement that references tables and row sets from one or more external OLE DB data sources". Oracle Corporation provides a more language-centric view in which distributed queries and distributed transactions form part of **distributed SQL.**

Architecture

A database user accesses the distributed database through:

Local applications
applications which do not require data from other sites.
Global applications
applications which do require data from other sites.

A **homogeneous distributed database** has identical software and hardware running all databases instances, and may appear through a single interface as if it were a single database. A **heterogeneous distributed database** may have different hardware, operating systems, database management systems, and even data models for different databases.

Homogeneous DDBMS

In a homogeneous distributed database all sites have identical software and are aware of each other and agree to cooperate in processing user requests. Each site surrenders part of its autonomy in terms of right to change schema or software. A homogeneous DDBMS appears to the user as a single system. The homogeneous system is much easier to design and manage. The following conditions must be satisfied for homogeneous database:

- The operating system used, at each location must be same or compatible.[*according to whom?*][*further explanation needed*]
- The data structures used at each location must be same or compatible.
- The database application (or DBMS) used at each location must be same or compatible.

Heterogeneous DDBMS

SAKI Institute of Science and Technology and English Language. In a heterogeneous distributed database different sites may use different schema and software. Difference in schema is a major problem for query processing and transaction processing. Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing. In heterogeneous systems, different nodes may have different hardware & software and data structures at various nodes or locations are also incompatible. Different computers and operating systems, database applications or data models may be used at each of the locations. For example, one location may have the latest relational database management technology, while another location may store data using conventional files or old version of database management system. Similarly, one location may have the Windows NT operating system, while another may have UNIX. Heterogeneous systems are usually used when individual sites use their own hardware and software. On heterogeneous system, translations are required to allow communication between different sites (or DBMS). In this system, the users must be able to make requests in a database language at their local sites. Usually the SQL database language is used for this purpose. If the hardware is different, then the translation is straightforward, in which computer codes and word-length is changed. The heterogeneous system is often not technically or economically feasible. In this system, a user at one location may be able to read but not update the data at another location.

Important considerations

Care with a distributed database must be taken to ensure the following:

•The distribution is transparent — users must be able to interact with the system as if it were one logical system. This applies to the system's performance, and methods of access among other things.

•Transactions are transparent — each transaction must maintain database integrity across multiple databases. Transactions must also be divided into sub-transactions, each sub-transaction affecting one database system.

There are two principal approaches to store a relation r in a distributed database system:

A) Replication
B) Fragmentation/Partitioning

A) Replication: In replication, the system maintains several identical replicas of the same relation r in different sites.

- Data is more available in this scheme.
- Parallelism is increased when read request is served.
- Increases overhead on update operations as each site containing the replica needed to be updated in order to maintain consistency.
- Multi-datacenter replication provides geographical diversity: http://basho.com/tag/multi-datacenter-replication/

B) Fragmentation: The relation r is fragmented into several relations r1, r2, r3....rn in such a way that the actual relation could be reconstructed from the fragments and then the fragments are scattered to different locations. There are basically two schemes of fragmentation:

- Horizontal fragmentation - splits the relation by assigning each tuple of r to one or more fragments.
- Vertical fragmentation - splits the relation by decomposing the schema R of relation r.

Advantages

•Management of distributed data with different levels of transparency like network transparency, fragmentation transparency, replication transparency, etc

•Increase reliability and availability

•Easier expansion

- Reflects organizational structure — database fragments potentially stored within the departments they relate to
- Local autonomy or site autonomy — a department can control the data about them (as they are the ones familiar with it)
- Protection of valuable data — if there were ever a catastrophic event such as a fire, all of the data would not be in one place, but distributed in multiple locations
- Improved performance — data is located near the site of greatest demand, and the database systems themselves are parallelized, allowing load on the databases to be balanced among servers. (A high load on one module of the database won't affect other modules of the database in a distributed database)
- Economics — it may cost less to create a network of smaller computers with the power of a single large computer
- Modularity — systems can be modified, added and removed from the distributed database without affecting other modules (systems)
- Reliable transactions - due to replication of the database
- Hardware, operating-system, network, fragmentation, DBMS, replication and location independence
- Continuous operation, even if some nodes go offline (depending on design)
- Distributed query processing can improve performance
- Distributed transaction management
- Single-site failure does not affect performance of system.
- All transactions follow A.C.I.D. property:
- A-atomicity, the transaction takes place as a whole or not at all
- C-consistency, maps one consistent DB state to another
- I-isolation, each transaction sees a consistent DB
- D-durability, the results of a transaction must survive system failures

The Merge Replication Method is popularly used to consolidate the data between databases.

Disadvantages

- Complexity — DBAs may have to do extra work to ensure that the distributed nature of the system is transparent. Extra work must also be done to maintain multiple disparate systems, instead of one big one. Extra database design work must also be done to account for the disconnected nature of the

database — for example, joins become prohibitively expensive when performed across multiple systems.

- Economics — increased complexity and a more extensive infrastructure means extra labour costs
- Security — remote database fragments must be secured, and they are not centralized so the remote sites must be secured as well. The infrastructure must also be secured (for example, by encrypting the network links between remote sites).
- Difficult to maintain integrity — but in a distributed database, enforcing integrity over a network may require too much of the network's resources to be feasible
- Inexperience — distributed databases are difficult to work with, and in such a young field there is not much readily available experience in "proper" practice
- Lack of standards — there are no tools or methodologies yet to help users convert a centralized DBMS into a distributed DBMS[*citation needed*]
- Database design more complex — besides of the normal difficulties, the design of a distributed database has to consider fragmentation of data, allocation of fragments to specific sites and data replication
- Additional software is required
- Operating system should support distributed environment
- Concurrency control poses a major issue. It can be solved by locking and timestamping.
- Distributed access to data
- Analysis of distributed data

1.8 Security considerations

Database security concerns the use of a broad range of information security controls to protect databases (potentially including the data, the database applications or stored functions, the database systems, the database servers and the associated network links) against compromises of their confidentiality, integrity and availability. It involves various types or categories of controls, such as technical, procedural/administrative and physical. *Database security* is a specialist topic within the broader realms

of computer security, information security and risk management.

Security risks to database systems include, for example:

- Unauthorized or unintended activity or misuse by authorized database users, database administrators, or network/systems managers, or by unauthorized users or hackers (e.g. inappropriate access to sensitive data, metadata or functions within databases, or inappropriate changes to the database programs, structures or security configurations);
- Malware infections causing incidents such as unauthorized access, leakage or disclosure of personal or proprietary data, deletion of or damage to the data or programs, interruption or denial of authorized access to the database, attacks on other systems and the unanticipated failure of database services;
- Overloads, performance constraints and capacity issues resulting in the inability of authorized users to use databases as intended;
- Physical damage to database servers caused by computer room fires or floods, overheating, lightning, accidental liquid spills, static discharge, electronic breakdowns/equipment failures and obsolescence;
- Design flaws and programming bugs in databases and the associated programs and systems, creating various security vulnerabilities (e.g. unauthorized privilege escalation), data loss/corruption, performance degradation etc.;
- Data corruption and/or loss caused by the entry of invalid data or commands, mistakes in database or system administration processes, sabotage/criminal damage etc.

Many layers and types of information security control are appropriate to databases, including:

- Access control
- Auditing
- Authentication
- Encryption
- Integrity controls
- Backups
- Application security

Traditionally databases have been largely secured against hackers through network security measures such as firewalls, and network-based intrusion detectionsystems. While network security controls remain valuable

in this regard, securing the database systems themselves, and the programs/functions and data within them, has arguably become more critical as networks are increasingly opened to wider access, in particular access from the Internet. Furthermore, system, program, function and data access controls, along with the associated user identification, authentication and rights management functions, have always been important to limit and in some cases log the activities of authorized users and administrators. In other words, these are complementary approaches to database security, working from both the outside-in and the inside-out as it were.

Many organizations develop their own "baseline" security standards and designs detailing basic security control measures for their database systems. These may reflect general information security requirements or obligations imposed by corporate information security policies and applicable laws and regulations (e.g. concerning privacy, financial management and reporting systems), along with generally-accepted good database security practices (such as appropriate hardening of the underlying systems) and perhaps security recommendations from the relevant database system and software vendors. The security designs for specific database systems typically specify further security administration and management functions (such as administration and reporting of user access rights, log management and analysis, database replication/synchronization and backups) along with various business-driven information security controls within the database programs and functions (e.g. data entry validation and audit trails). Furthermore, various security-related activities (manual controls) are normally incorporated into the procedures, guidelines etc. relating to the design, development, configuration, use, management and maintenance of databases.

Vulnerability Assessments and Compliance

One technique for evaluating database security involves performing vulnerability assessments or penetration tests against the database. Testers attempt to find security vulnerabilities that could be used to defeat or bypass security controls, break into the database, compromise the system etc. Database administrators or information security administrators may for example use automated vulnerability scans to search out misconfiguration of controls within the layers mentioned above along with known vulnerabilities within the database software. The results of such scans are used to harden the database (improve the security controls) and close off the specific vulnerabilities identified, but unfortunately other vulnerabilities typically remain unrecognized and unaddressed.

A program of continual monitoring for compliance with database security standards is another important task for mission critical database environments. Two crucial aspects of database security compliance include patch management and the review and management of permissions (especially public) granted to objects within the database. Database objects may include table or other objects listed in the Table link. The permissions granted for SQL language commands on objects are considered in this process. One should note that compliance monitoring is similar to vulnerability assessment with the key difference that the results of vulnerability assessments generally drive the security standards that lead to the continuous monitoring program. Essentially, vulnerability assessment is a preliminary procedure to determine risk where a compliance program is the process of on-going risk assessment.

The compliance program should take into consideration any dependencies at the application software level as changes at the database level may have effects on the application software or the application server. In direct relation to this topic is that of application security.

Abstraction

Application level authentication and authorization mechanisms should be considered as an effective means of providing abstraction from the database layer. The primary benefit of abstraction is that of a single sign-on capability across multiple databases and database platforms. A Single sign-on system should store the database user's credentials (login id and password), and authenticate to the database on behalf of the user.

Database activity monitoring (DAM)

Another security layer of a more sophisticated nature includes real-time database activity monitoring, either by analyzing protocol traffic (SQL) over the network, or by observing local database activity on each server using software agents, or both. Use of agents or native logging is required to capture activities executed on the database server, which typically include the activities of the database administrator. Agents allow this information to be captured in a fashion that can not be disabled by the database administrator, who has the ability to disable or modify native audit logs..

Analysis can be performed to identify known exploits or policy breaches, or baselines can be captured over time to build a normal pattern used for detection of anomalous activity that could be indicative of intrusion. These systems can provide a comprehensive Database audit trail in addition to the intrusion detection mechanisms, and some systems can also provide protection by terminating user sessions and/or quarantining users demonstrating suspicious behavior. Some systems are designed to support separation of

duties (SOD), which is a typical requirement of auditors. SOD requires that the database administrators who are typically monitored as part of the DAM, not be able to disable or alter the DAM functionality. This requires the DAM audit trail to be securely stored in a separate system not administered by the database administration group.

Native Audit

In addition to using external tools for monitoring or auditing, native database audit capabilities are also available for many database platforms. The native audit trails are extracted on a regular basis and transferred to a designated security system where the database administrators do not have access. This ensures a certain level of segregation of duties that may provide evidence the native audit trails were not modified by authenticated administrators. Turning on native impacts the performance of the server. Generally, the native audit trails of databases do not provide sufficient controls to enforce separation of duties; therefore, the network and/or kernel module level host based monitoring capabilities provides a higher degree of confidence for forsenics and preservation of evidence.

Process and Procedures

A *database security* program should include the regular review of permissions granted to individually owned accounts and accounts used by automated processes. The accounts used by automated processes should have appropriate controls around password storage such as sufficient encryption and access controls to reduce the risk of compromise. For individual accounts, a two-factor authentication system should be considered in a database environment where the risk is commensurate with the expenditure for such an authentication system.

In conjunction with a sound *database security* program, an appropriate disaster recovery program should exist to ensure that service is not interrupted during a security incident or any other incident that results in an outage of the primary database environment. An example is that of replication for the primary databases to sites located in different geographical regions.

After an incident occurs, the usage of database forensics should be employed to determine the scope of the breach, and to identify appropriate changes to systems and/or processes to prevent similar incidents in the future.

Security considerations for a distributed relational database

Part of planning for a distributed relational database involves the decisions you must make about securing distributed data.

These decisions include:

- What systems should be made accessible to users in other locations and which users in other locations should have access to those systems.
- How tightly controlled access to those systems should be. For example, should a user password be required when a conversation is started by a remote user?
- Is it required that passwords flow over the wire in encrypted form?
- Is it required that a user profile under which a client job runs be mapped to a different user identification or password based on the name of the relational database to which you are connecting?
- What data should be made accessible to users in other locations and which users in other locations should have access to that data.
- What actions those users should be allowed to take on the data.
- Whether authorization to data should be centrally controlled or locally controlled.
- If special precautions should be taken because multiple systems are being linked. For example, should name translation be used?

When making the previous decisions, consider the following items when choosing locations:

- Physical protection. For example, a location might offer a room with restricted access.
- Level of system security. The level of system security often differs between locations. The security level of the distributed database is no greater than the lowest level of security used in the network.

  All systems connected by Advanced Program-to-Program Communication (APPC) can do the following things:

- If both systems are System i® products, communicate passwords in encrypted form.
- When one system receives a request to communicate with another system in the network, verify that the requesting system is actually "who it says it is" and that it is authorized to communicate with the receiving system.

All systems can do the following things:

- Pass a user's identification and password from the local system to the remote system for verification before any remote data access is allowed.
- Grant and revoke privileges to access and manipulate SQL objects such as tables and views.

  The i5/OS® operating system includes security audit functions that you can use to track unauthorized attempts to access data, as well as to track other events pertinent to security. The system also provides a function that can prevent all distributed database access from remote systems.

- Security-related costs. When considering the cost of security, consider both the cost of buying security-related products and the price of your information staff's time to perform the following activities:
- Maintain identification of remote-data-accessing users at both local and remote systems.
- Coordinate auditing functions between sites.

Unit 2: The Extended Entity Relationship Model and Object Model:

2.1 The ER model revisited

In software engineering, an **entity–relationship model** (**ER model**) is a data modelfor describing a database in an abstract way. This article refers to the techniques proposed in Peter Chen's 1976 paper. However, variants of the idea existed previously, and have been devised subsequently such as supertype and subtype data entities and commonality relationships.

An ER model is an abstract way of describing a database. In the case of a relational database, which stores data in tables, some of the data in these tables point to data in other tables - for instance, your entry in the database could point to several entries for each of the phone numbers that are yours. The ER model would say that you are an entity, and each phone number is an entity, and the relationship between you and the phone numbers is 'has a phone number'. Diagrams created to design these entities and relationships are called entity–relationship diagrams or ER diagrams.

Using the three schema approach to software engineering, there are three levels of ER models that may be developed.

Conceptual data model

This is the highest level ER model in that it contains the least granular detail but establishes the overall scope of what is to be included within the model set. The conceptual ER model normally defines master reference data entities that are commonly used by the organization. Developing an enterprise-wide conceptual ER model is useful to support documenting the data architecture for an organization.

A conceptual ER model may be used as the foundation for one or more *logical data models* (see below). The purpose of the conceptual ER model is then to establish structural metadata commonality for the master data entities between the set of logical ER models. The conceptual data model may be used to form commonality relationships between ER models as a basis for data model integration.

Logical data model

A logical ER model does not require a conceptual ER model, especially if the scope of the logical ER model

includes only the development of a distinct information system. The logical ER model contains more detail than the conceptual ER model. In addition to master data entities, operational and transactional data entities are now defined. The details of each data entity are developed and the entity relationships between these data entities are established. The logical ER model is however developed independent of technology into which it will be implemented.

Physical model

One or more physical ER models may be developed from each logical ER model. The physical ER model is normally developed to be instantiated as a database. Therefore, each physical ER model must contain enough detail to produce a database and each physical ER model is technology dependent since each database management system is somewhat different.

The physical model is normally forward engineered to instantiate the structural metadata into a database management system as relational database objects such as database tables, database indexes such as unique key indexes, and database constraints such as a foreign key constraint or a commonality constraint. The ER model is also normally used to design modifications to the relational database objects and to maintain the structural metadata of the database.

The first stage of information system design uses these models during the requirements analysis to describe information needs or the type of information that is to be stored in a database. The data modeling technique can be used to describe any ontology (i.e. an overview and classifications of used terms and their relationships) for a certain area of interest. In the case of the design of an information system that is based on a database, the conceptual data model is, at a later stage (usually called logical design), mapped to a logical data model, such as the relational model; this in turn is mapped to a physical model during physical design. Note that sometimes, both of these phases are referred to as "physical design". It is also used in database management system.

Diagramming conventions

Various methods of representing the same one to many relationship. In each case, the diagram shows the relationship between a person and a place of birth: each person must have been born at one, and only one, location, but each location may have had zero or more people born at it.

Two related entities shown using Crow's Foot notation. In this example, an optional relationship is shown

between Artist and Song; the symbols closest to the song entity represents "zero, one, or many", whereas a song has "one and only one" Artist. The former is therefore read as, an Artist (can) perform(s) "zero, one, or many" song(s).

Chen's notation for entity–relationship modeling uses rectangles to represent entity sets, and diamonds to represent relationships appropriate for first-class objects: they can have attributes and relationships of their own. If an entity set participates in a relationship set, they are connected with a line.

Attributes are drawn as ovals and are connected with a line to exactly one entity or relationship set.

Cardinality constraints are expressed as follows:

- a double line indicates a *participation constraint*, totality or surjectivity: all entities in the entity set must participate in *at least one* relationship in the relationship set;
- an arrow from entity set to relationship set indicates a key constraint, i.e. injectivity: each entity of the entity set can participate in *at most one* relationship in the relationship set;
- a thick line indicates both, i.e. bijectivity: each entity in the entity set is involved in *exactly one* relationship.
- an underlined name of an attribute indicates that it is a key: two different entities or relationships with this attribute always have different values for this attribute.

Attributes are often omitted as they can clutter up a diagram; other diagram techniques often list entity attributes within the rectangles drawn for entity sets.

Limitations

- ER models assume information content that can readily be represented in a relational database. They describe only a relational structure for this information.
- They are inadequate for systems in which the information cannot readily be represented in relational form, such as with semi-structured data.
- For many systems, the possible changes to the information contained are nontrivial and important enough to warrant explicit specification.
- Some[*who?*] authors have extended ER modeling with constructs to represent change, an approach supported by the original author; an example is Anchor Modeling. An alternative is to model change

separately, using a process modeling technique. Additional techniques can be used for other aspects of systems. For instance, ER models roughly correspond to just 1 of the 14 different modeling techniques offered by UML.

- ER modeling is aimed at specifying information from scratch. This suits the design of new, standalone information systems, but is of less help in integrating pre-existing information sources that already define their own data representations in detail.
- Even where it is suitable in principle, ER modeling is rarely used as a separate activity. One reason for this is today's abundance of tools to support diagramming and other design support directly on relational database management systems. These tools can readily extract database diagrams that are very close to ER diagrams from existing databases, and they provide alternative views on the information contained in such diagrams.
- In a survey, Brodie and Liu could not find a single instance of entity–relationship modeling inside a sample of ten Fortune 100 companies. Badia and Lemire blame this lack of use on the lack of guidance but also on the lack of benefits, such as lack of support for data integration.
- The enhanced entity–relationship model (EER modeling) introduces several concepts which are not present in ER modeling, which are closely related to object-oriented design, like is-a relationships.
- For modeling temporal databases, numerous ER extensions have been considered. Similarly, the ER model was found unsuitable for multidimensional databases(used in OLAP applications); no dominant conceptual model has emerged in this field yet, although they generally revolve around the concept of OLAP cube (also known as *data cube* within the field).

Enhanced entity–relationship model

The **enhanced entity–relationship (EER)** model (or **extended entity-relationship** model) in computer science is a high-level or conceptual data model incorporating extensions to the original entity–relationship (ER) model, used in the design of databases.

It was developed to reflect more precisely the properties and constraints that are found in more complex databases, such as in engineering design and manufacturing (CAD/CAM), telecommunications, complex software systems and geographic information systems (GIS).

The EER model

The EER model includes all of the concepts introduced by the ER model. Additionally it includes the concepts of a subclass and superclass (Is-a), along with the concepts of specialization and generalization. Furthermore, it introduces the concept of a union type or category, which is used to represent a collection of objects that is the union of objects of different entity types.

Subclass and superclass

Entity type Y is a subtype (subclass) of an entity type X if and only if every Y is necessarily an X. A subclass entity inherits all attributes and relationships of its superclass entity. A subclass entity may have its own specific attributes and relationships (together with all the attributes and relationships it inherits from the superclass. One of the most common superclass examples is a vehicle with subclasses of Car and Truck. There are a number of common attributes between a car and a truck, which would be part of the Superclass, while the attributes specific to a car or a truck (such as max payload, truck type...) would make up two subclasses.

Software tools for the EER model

The MySQL Workbench offers creating, editing and exporting EER Models. Exporting to PNG and PDF allows easy sharing for presentations.

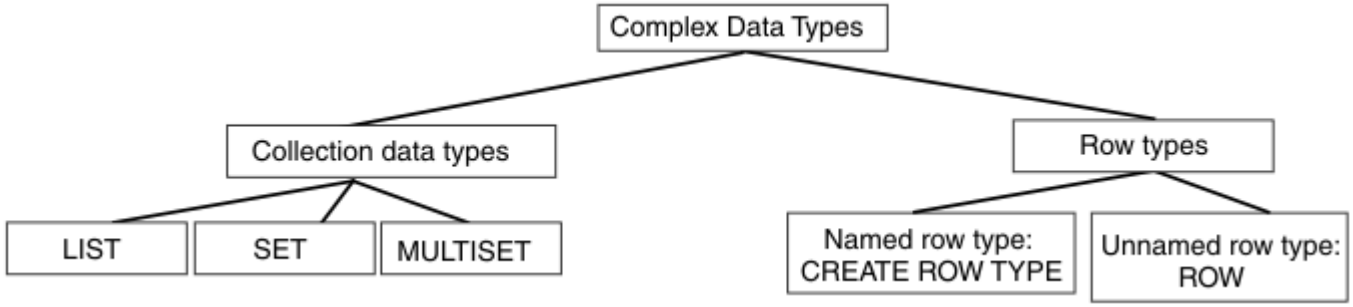
2.2 Motivation for complex data types

Complex Data Types

A complex data type is usually a composite of other existing data types. For example, you might create a complex data type whose components include built-in types, opaque types, distinct types, or other complex types. An important advantage that complex data types have over user-defined types is that users can access and manipulate the individual components of a complex data type.

In contrast, built-in types and user-defined types are self-contained (encapsulated) data types. Consequently, the only way to access the component values of an opaque data type is through functions that you define on the opaque type.

Figure shows the complex data types that Dynamic Server supports and the syntax that you use to create the complex data types.



The complex data types that Figure illustrates provide the following extended data type support:

- **Collection types.** You can use a collection type whenever you need to store and manipulate collections of data within a table cell. You can assign collection types to columns.
- **Row types**. A row type typically contains multiple fields. When you want to store more than one kind of data in a column or variable, you can create a row type. Row types come in two kinds: named row types and unnamed row types. You can assign an unnamed row type to columns and variables. You can assign a named row type to columns, variables, tables, or views. When you assign a named row type to a table, the table is a *typed table*. A primary advantage of typed tables is that they can be used to define an inheritance hierarchy.

2.3 User defined abstract data types and structured types

By using *abstract data types*, which are user-defined types, together with various routines, you can uniquely define and use data with complex structures and perform operations on such data. When you define a column as having an abstract data type, you can conceptualize and model its data based on object-oriented concepts. In addition, by applying object-oriented software development techniques, you can reduce the workload for database design, UAP development, and maintenance.

An abstract data type can be defined in a database using the CREATE TYPE definition SQL shown as follows.

```
CREATE TYPE t_employee (
    Name        CHAR(16),
    Sex         CHAR(1),
    employment_date   DATE,
    position    CHAR(10),
    id_photo    BLOB(64K),
    salary      INTEGER,
    ....
    FUNCTION    service-years ( p t_employee )
      RETURNS  INTEGER
     BEGIN
      DECLARE service_years INTERVAL YEAR TO DAY;
      SET service_years = CURRENT_DATE -
p..employment_date;
      RETURN YEAR(service_years);
     END,
    ....
 )
```

In this way, the user can use an abstract data type to define new data types by specifying attributes and operations.

Structured data type

A structured data type is a user-defined data type with elements that are not atomic; rather, they are divisible and can be used either separately or as a single unit, as appropriate.

Structured data types exhibit a behavior known as *inheritance*. A structured type can have *subtypes*, other structured types that reuse all of its attributes and contain their own specific attributes. The type from which a subtype inherits attributes is known as its *supertype*. A *type hierarchy* is a set of subtypes that are based on the same supertype; the pre-eminent supertype in a hierarchy is known as the *root type* of the hierarchy.

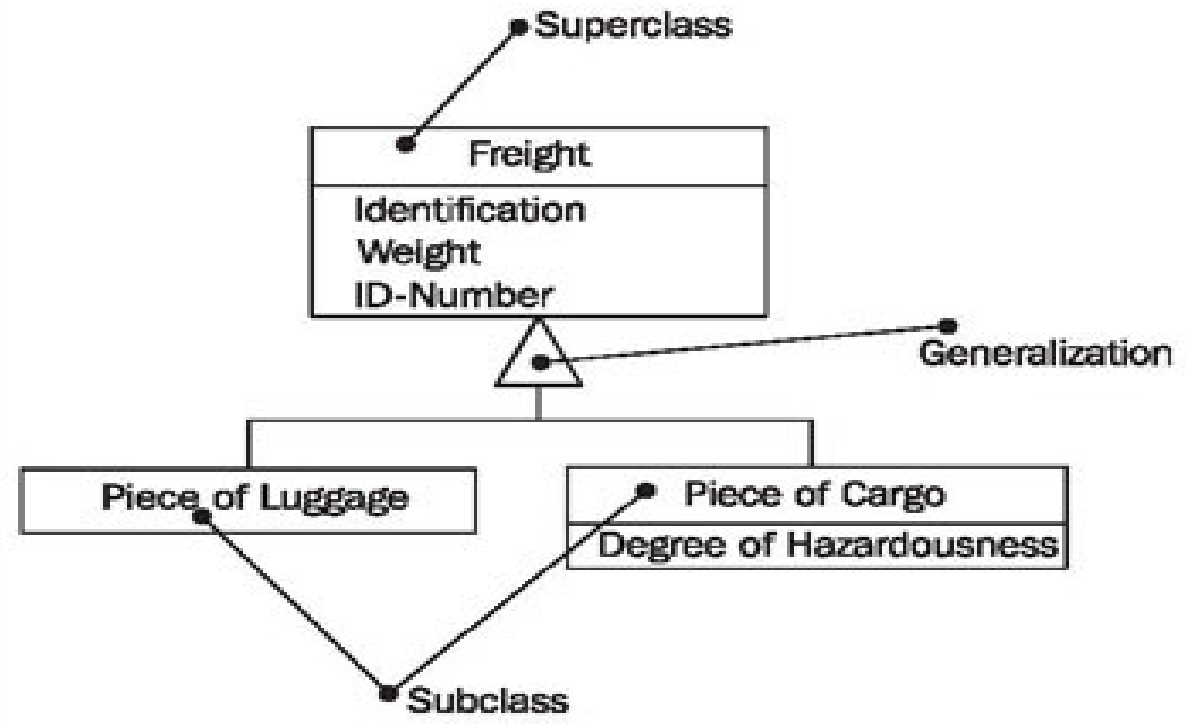
Use the CREATE TYPE statement to create a structured type, and use the DROP statement to delete a structured type.

- The *constructor function* has the same name as the structured type with which it is associated. The constructor function has no parameters and returns an instance of the type with all of its attributes set to null values.
- A *mutator method* exists for each attribute of a structured type. When you invoke a mutator method on an instance of a structured type and specify a new value for its associated attribute, the method returns a new instance with the attribute updated to the new value.
- An *observer method* exists for each attribute of a structured type. When you invoke an observer method on an instance of a structured type, the method returns the value of the attribute for that instance.

2.4 Subclasses, Super classes

Terms such as superclass, subclass, or inheritance come to mind when thinking about the object-oriented approach. These concepts are very important when dealing with object-oriented programming languages such as Java, Smalltalk, or C++. For modeling classes that illustrate technical concepts they are secondary. The reason for this is that modeling relevant objects or ideas from the real world gives little opportunity for using inheritance (compare the class diagram of our case study). Nevertheless, we would like to further introduce these terms at this point in Figure 4.26:

Figure 4.26 Notation of generalization

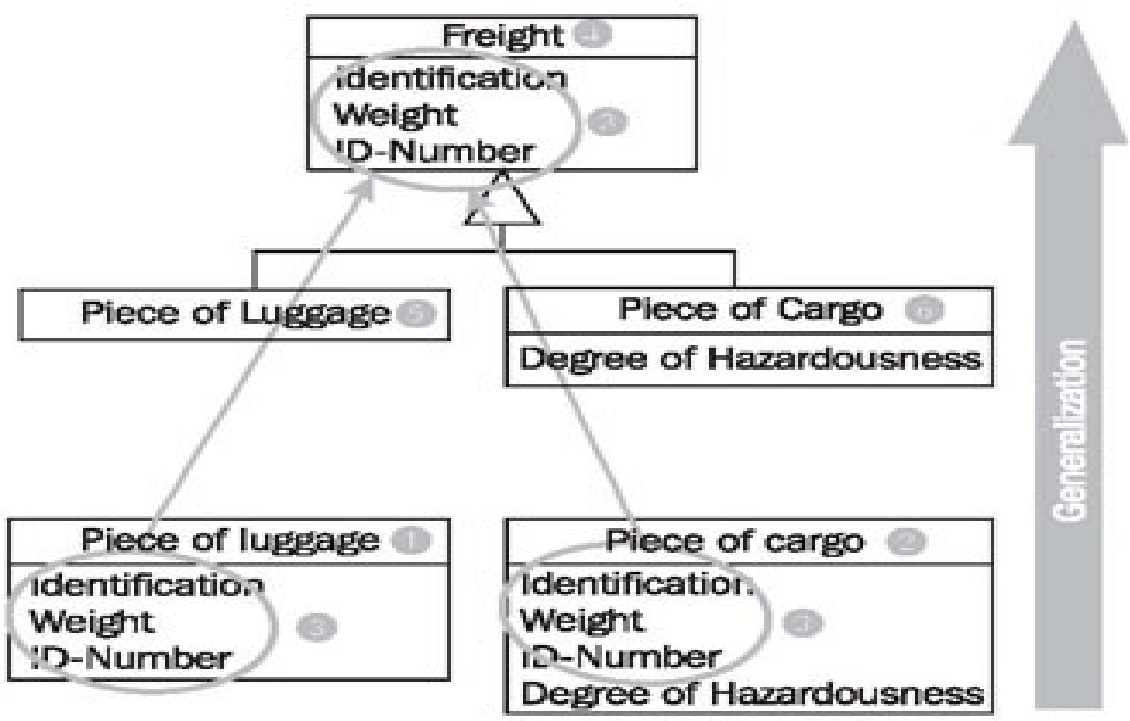


Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods.

In Figure 4.27, the classes *Piece of Luggage* (1) and *Piece of Cargo* (2) partially share the same attributes. From a domain perspective, the two classes are also very similar. During generalization, the shared characteristics (3) are combined and used to create a new superclass *Freight* (4). *Piece of Luggage* (5) and*Piece of Cargo* (6) become subclasses of the class *Freight*.
05
The shared attributes (3) are only listed in the superclass, but also apply to the two subclasses, even though they are not listed there.

Figure 4.27 Example of generalization

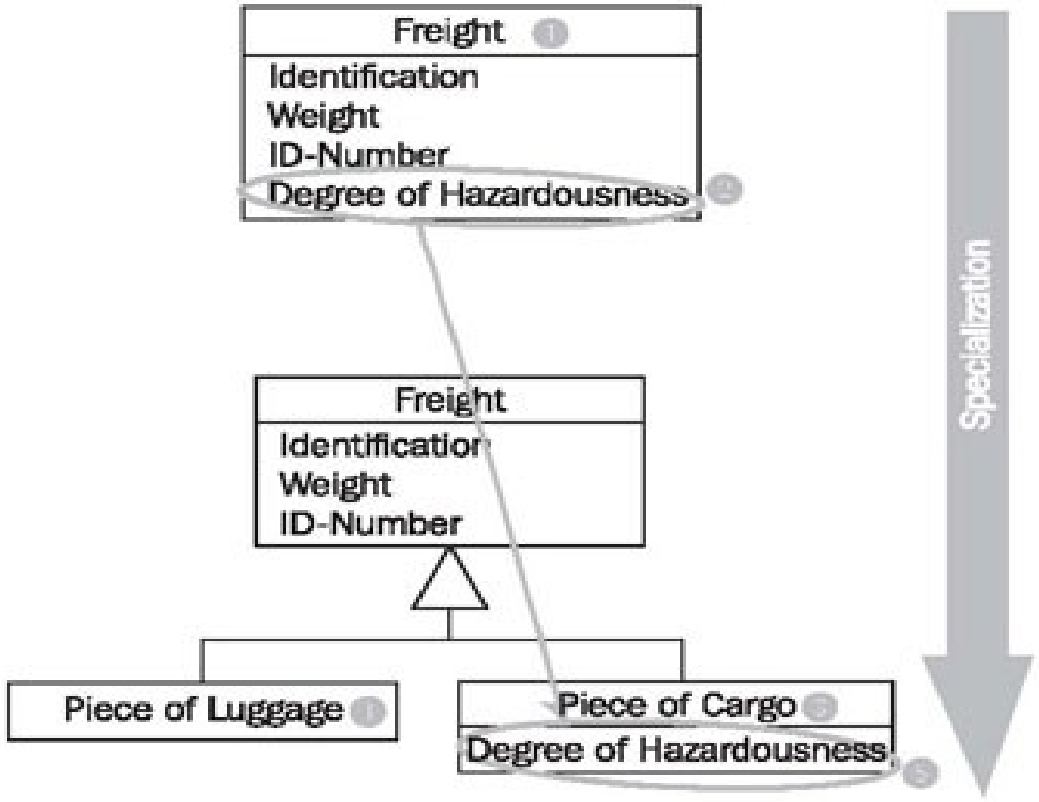
Consider whether some of the classes that you found could be generalized.



In contrast to generalization, ***specialization*** means creating new subclasses from an existing class. If it turns out that certain attributes, associations, or methods only apply to some of the objects of the class, a subclass can be created. The most inclusive class in a generalization/specialization is called the superclass and is generally located at the top of the diagram. The more specific classes are called subclasses and are generally placed below the superclass.

In Figure 4.28, the class *Freight* (1) has the attribute *Degree of Hazardousness* (2), which is needed only for cargo, but not for passenger luggage. Additionally (not visible in Figure 4.28), only passenger luggage has a connection to a coupon. Obviously, here two similar but different domain concepts are combined into one class. Through specialization the two special cases of freights are formed: *Piece of Cargo* (3) and *Piece of Luggage* (4). The attribute *Degree of Hazardousness* (5) is placed where it belongs—in *Piece of Cargo*. The attributes of the class *Freight* (1) also apply to the two subclasses *Piece of Cargo* (3) and *Piece of Luggage* (4):

Figure 4.28 Example of specialization



Consider whether some of the classes that you found could be specialized.

So much for the mechanism. However, the domain meanig of the relationship between superclass and subclass is much more important. These rules apply to this relationship:

- All statements that are made about a superclass also apply to all subclasses. We say that subclasses "*inherit*" attributes, associations, and operations from the superclass. For example: If the superclass *Freight* has an attribute *Weight*, then the subclass piece of luggage also has an attribute Weight, even though this attribute is not listed in the subclass *Piece of Luggage*.
- Anything that can be done with an object of the superclass can also be done with an object of the subclass. For example: If freight can be loaded, pieces of luggage can also be loaded.
- In the terminology of the system that is being modeled, a subclass has to be a special form of the superclass. For example: A piece of luggage is a special case of freight. The counter-example to this is: A flight is not a special case of a flight number.

Difference

generalization and specialization are important relationships that exist betweena higher level entity set and one or more lower level entity sets.

1. generalization is the result of taking the union of two or more lower level entity sets to produce a higher level entity sets.

specialization is the results of taking subsets of a higher level entity set to form a lower level entity sets.

2. In generalization,each higher level entity must also be a lower level entity.

In specialization,some higher level entities may not have lower-level entity sets at all.

3. Specialization is a **Top Down** process where as Generalization is **Bottom Up** process.

2.5 Inheritance

Single table inheritance is a way to emulate object-oriented inheritance in a relational database. When mapping from a database table to an object in an object-oriented language, a field in the database identifies what class in the hierarchy the object belongs to. In Ruby on Rails the field in the table called 'type' identifies the name of the class. In .NET Framework, it is called the Discriminator column.

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let's create two tables: A table cities and a table capitals. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you're really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
  name       text,
  population real,
  altitude   int,    -- (in ft)
  state      char(2)
```

```
);
CREATE TABLE non_capitals (
  name       text,
  population real,
  altitude   int     -- (in ft)
);
CREATE VIEW cities AS
  SELECT name, population, altitude FROM capitals
    UNION
  SELECT name, population, altitude FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, to name one thing.

A better solution is this:

```
CREATE TABLE cities (
  name       text,
  population real,
  altitude   int     -- (in ft)
);
CREATE TABLE capitals (
  state      char(2)
) INHERITS (cities);
```

In this case, a row of capitals *inherits* all columns (name, population, and altitude) from its *parent*, cities. The type of the column name is text, a native PostgreSQL type for variable length character strings. State capitals have an extra column, state, that shows their state. In PostgreSQL, a table can inherit from zero

or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an altitude over 500 ft.:

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

which returns:

```
  name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an altitude of 500 ft. or higher:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;
  name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953          (2 rows)
```

Here the ONLY before cities indicates that the query should be run over only the cities table, and not tables below cities in the inheritance hierarchy. Many of the commands that we have already discussed -- SELECT, UPDATE, and DELETE -- support this ONLY notation.

2.6 Specialization and Generalization

specialization, generalization and aggregation are the feature of extended E-R

1. Generalization:

A generalization hierarchy is a form of abstraction that specifies that two or more entities that share common attributes can be generalized into a higher-level entity type called a super type or generic entity. The lower level of entities becomes the subtypes, or categories, to the super type. Subtypes are dependent entities.

Generalization is used to emphasize the similarities among lower-level entity set and to hide differences. It makes ER diagram simpler because shared attributes are not repeated. Generalization is denoted through a triangular component labeled ‘IS A”.

2. Specialization:

Specialization is the process of taking subsets of the higher-level entity set to form lower-level entity sets. It is a process of defining a set of subclasses of an entity type, which is called the super class of the specialization. The process of defining subclass is based on the basis of some distinguish characteristics of the entities in the super class.

3. Aggregation:

Aggregation is the process of compiling information on an object, thereby abstracting a higher-level object. One limitation of the E-R model is that it cannot express relationships among relationships.

To illustrate the need for such a construct, consider the ternary relationship **works-on**, between **employee**, **branch** and **job**. The best way to model a situation like this is by the use of aggregation. Thus the relationship set **works-on** relating the entity sets **employee**, **branch** and **job** is a higher-level entity set. Such an entity set is treated in the same manner, as is any other entity set. We can then create a binary relationship **manages** between **works-on** and **manager** to represent who manages what task.

2.7 Constraints and characteristics of specialization and Generalization

2.8 Relationship types of degree higher than two

Relationships of Higher Degree

Relationship types of degree 2 are called binary. Relationship types of degree 3 are called ternary and of degree n are called n-ary. In general, an n-ary relationship is not equivalent to n binary relationships. Constraints are harder to specify for higher-degree relationships (n > 2) than for binary relationships.

In general, 3 binary relationships can represent different information than a single ternary relationship (see Figure 3.17a and b on next slide). If needed, the binary and n-ary relationships can all be included in the schema design (see Figure 3.17a and b, where all relationships convey different meanings). In some cases, a ternary relationship can be represented as a weak entity if the data model allows a weak entity type to have multiple identifying relationships (and hence multiple owner entity types).

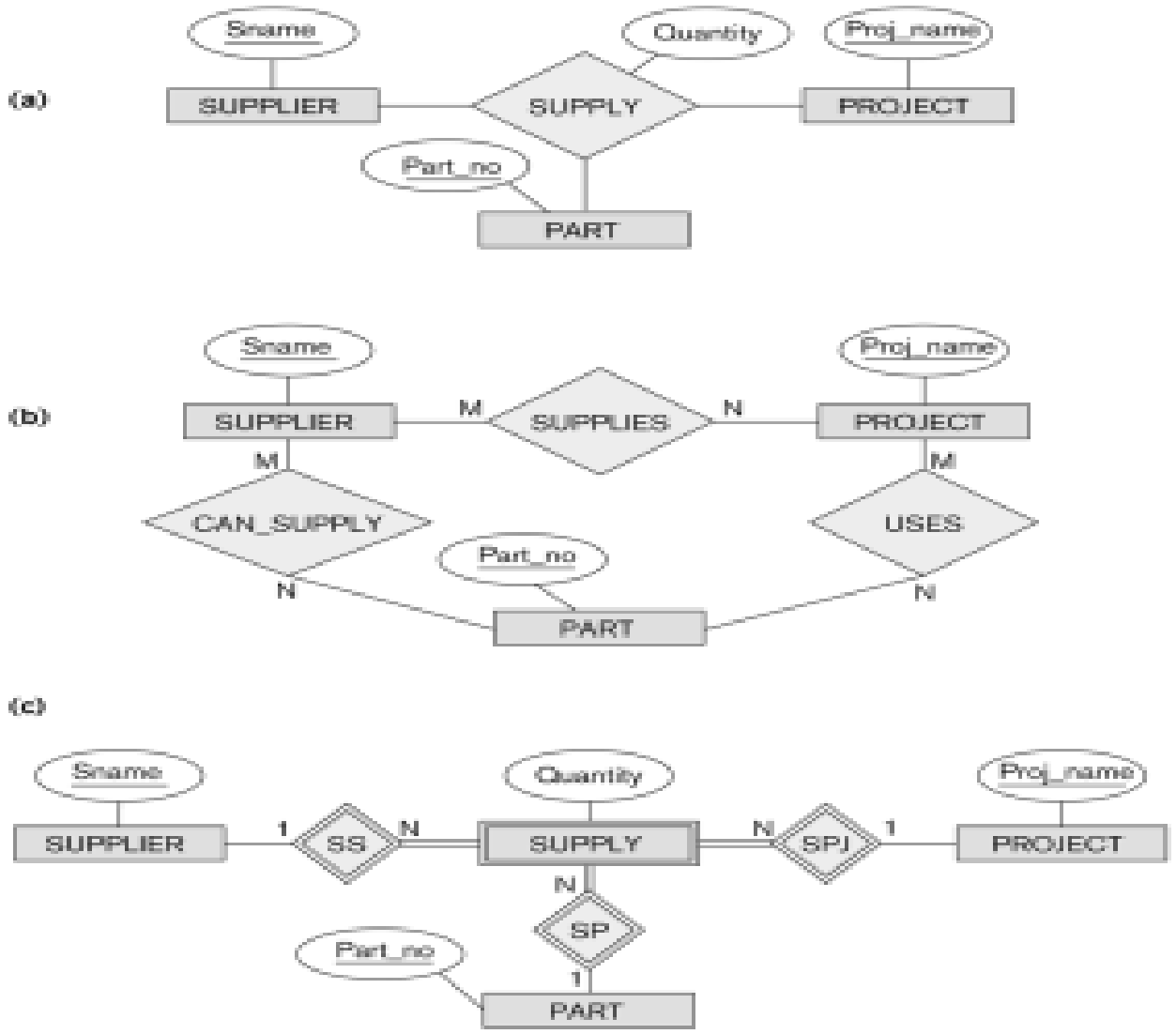
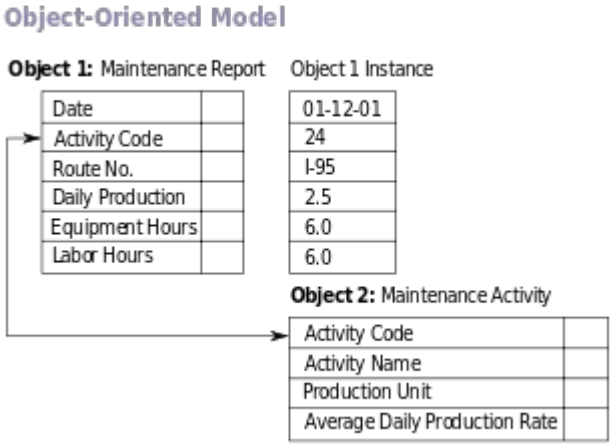


Figure 3.17
Ternary relationship types. (a) The SUPPLY relationship. (b) Three binary relationships not equivalent to SUPPLY. (c) SUPPLY represented as a weak entity type.

Unit 3: Emerging Database Management System Technologies

3.1 Object Oriented database concepts

An **object database** (also **object-oriented database management system**) is a database management system in which information is represented in the form of **objects** as used in object-oriented programming. Object databases are different from relational databases which are table-oriented. Object-relational databases are a hybrid of both approaches.



Object databases have been considered since the early 1980s.

Object databases based on persistent programming acquired a niche in application areas such as engineering and spatial databases, telecommunications, and scientific areas such as high energy physics and molecular biology.

Another group of object databases focuses on embedded use in devices, packaged software, and real-time systems.

Comparison with RDBMSs

An object database stores complex data and relationships between data directly, without mapping to relational rows and columns, and this makes them suitable for applications dealing with very complex data. Objects have a many to many relationship and are accessed by the use of pointers. Pointers are linked to objects to establish relationships. Another benefit of an OODBMS is that it can be programmed with small procedural differences without affecting the entire system. This is most helpful for those organizations that have data relationships that are not entirely clear or need to change these relations to satisfy the new business requirements.

Potential advantages:

- •Objects don't require assembly and disassembly saving coding time and execution time to assemble or disassemble objects.
- •Reduced paging.
- •Easier navigation.
- •Better concurrency control - a hierarchy of objects may be locked.
- •Data model is based on the real world.
- •Works well for distributed architectures.
- •Less code required when applications are object oriented.

Potential disadvantages:

- •Lower efficiency when data is simple and relationships are simple.
- •Relational tables are simpler.
- •Late binding may slow access speed.
- •More user tools exist for RDBMS.
- •Standards for RDBMS are more stable.
- •Support for RDBMS is more certain and change is less likely to be required.

3.2 Object Relational database concepts

An **object-relational database** (**ORD**), or **object-relational database management system** (**ORDBMS**), is a database management system (DBMS) similar to arelational database, but with an object-oriented database model: objects, classes and inheritance are directly supported in database schemas and in the query language. In addition, just as with pure relational systems, it supports extension of the data model with custom data-types and methods.

An object-relational database can be said to provide a middle ground between relational databases and*object-oriented databases* (OODBMS). In object-relational databases, the approach is essentially that of

relational databases: the data resides in the database and is manipulated collectively with queries in a query language; at the other extreme are OODBMSes in which the database is essentially a persistent object store for software written in an object-oriented programming language, with a programming API for storing and retrieving objects, and little or no specific support for querying.

Comparison to RDBMS

An RDBMS might commonly involve SQL statements such as these:

```
CREATE TABLE Customers  (
   Id         CHAR(12)   NOT NULL PRIMARY KEY,
   Surname    VARCHAR(32) NOT NULL,
   FirstName  VARCHAR(32) NOT NULL,
   DOB        DATE       NOT NULL
);
SELECT InitCap(Surname) || ', ' || InitCap(FirstName)
  FROM Customers
 WHERE MONTH(DOB) = MONTH(getdate())
   AND DAY(DOB) = DAY(getdate())
```

Most current SQL databases allow the crafting of custom functions, which would allow the query to appear as:

```
SELECT Formal(Id)
  FROM Customers
 WHERE Birthday(DOB) = Today()
```

In an object-relational database, one might see something like this, with user-defined data-types and expressions such as BirthDay():

```
CREATE TABLE Customers (
   Id         Cust_Id    NOT NULL  PRIMARY KEY,
   Name       PersonName  NOT NULL,
   DOB        DATE       NOT NULL
);
SELECT Formal( C.Id )
  FROM Customers C
```

```
   WHERE BirthDay ( C.DOB ) = TODAY;
```

The object-relational model can offer another advantage in that the database can make use of the relationships between data to easily collect related records. In an address book application, an additional table would be added to the ones above to hold zero or more addresses for each customer. Using a traditional RDBMS, collecting information for both the user and their address requires a "join":

```
    SELECT InitCap(C.Surname) || ', ' || InitCap(C.FirstName), A.city
     FROM Customers C JOIN Addresses A ON A.Cust_Id=C.Id -- the join
    WHERE A.city="New York"
```

The same query in an object-relational database appears more simply:

```
   SELECT Formal( C.Name )
    FROM Customers C
   WHERE C.address.city="New York" -- the linkage is 'understood' by the ORDB
```

3.3 Active database concepts

An **active database** is a database that includes an event-driven architecture (often in the form of ECA rules) which can respond to conditions both inside and outside the database. Possible uses include security monitoring, alerting, statistics gathering and authorization.

Most modern relational databases include active database features in the form of database triggers.

(more on slides.....)

3.4 Temporal database concepts

Non-Temporal Databases

Commercial database management systems (DBMS) such as Oracle, Sybase, Informix and O2 allow the storage of huge amounts of data. This data is usually considered to be valid now. Past or future data is not stored. Past data refers to data which was stored in the database at an earlier time instant and which might

has been modified or deleted in the meantime. Past data usually is overwritten with new (updated) data. Future data refers to data which is considered to be valid at a future time instant (but not now).

A DBMS stores the data in a well-defined format. A relational DBMS, for example, stores data in tables (also called relations). Thus, a relational database actually contains a set of tables. Each table contains rows (tuples) and columns (attributes). A row contains data about a specific entity, for example, an employee. Each column specifies a certain property of these entities, for example, the employee's name, salary etc. The following table stores data about employees:

EmpID	Name	Department	Salary
10	John	Sales	12000
12	George	Research	10500
13	Ringo	Sales	15500

Object-oriented DBMS store data about entities in objects. So each employee is actually an object. The type of an object specifies the properties the object has. An employee object thus has properties such as a name, a salary etc. Sets of objects of the same type are called collections. Thus - in an object-oriented DBMS - a database contains a set of collections.

Temporal Databases

Temporal data strored in a temporal database is different from the data stored in non-temporal database in that a time period attached to the data expresses when it was valid or stored in the database. As mentioned above, conventional databases consider the data stored in it to be valid at time instant now, they do not keep track of past or future database states. By attaching a time period to the data, it becomes possible to store different database states.

A first step towards a temporal database thus is to timestamp the data. This allows the distinction of different database states. One approach is that a temporal database may timestamp entities with time periods. Another approach is the timestamping of the property values of the entities. In the relational data model, tuples are timestamped, where as in object-oriented data models, objects and/or attribute values may be timestamped.

What time period do we store in these timestamps? As we mentioned already, there are mainly two different notions of time which are relevant for temporal databases. One is called the **valid time**, the other one is the**transaction time**. Valid time denotes the time period during which a fact is true with respect to the real world. Transaction time is the time period during which a fact is stored in the database. Note that these two time periods do not have to be the same for a single fact. Imagine that we come up with a temporal database storing data about the 18th century. The valid time of these facts is somewhere between 1700 and 1799, where as the transaction time starts when we insert the facts into the database, for example, January 21, 1998.

Assume we would like to store data about our employees with respect to the real world. Then, the following table could result:

EmpID	Name	Department	Salary	ValidTimeStart	ValidTimeEnd
10	John	Research	11000	1985	1990
10	John	Sales	11000	1990	1993
10	John	Sales	12000	1993	INF
11	Paul	Research	10000	1988	1995

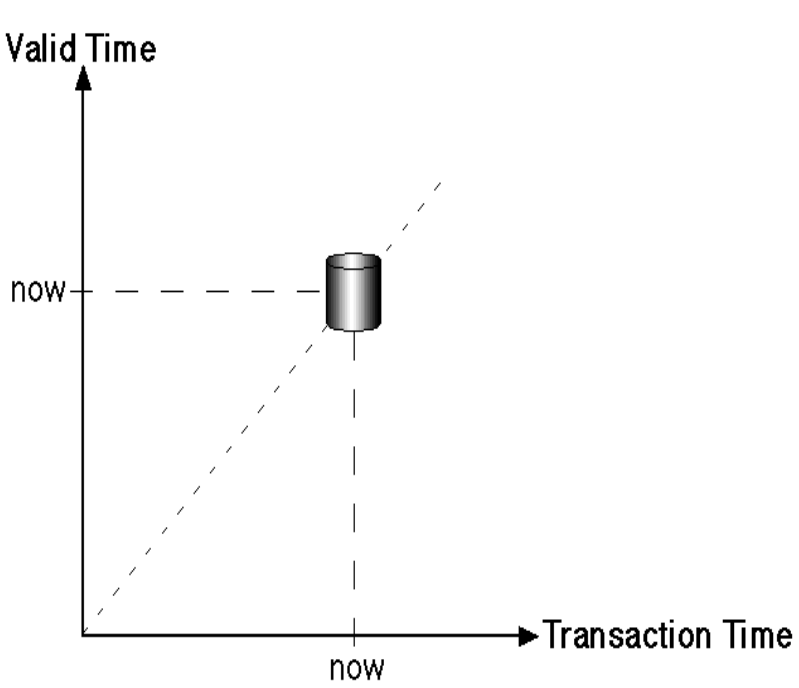
12	**George**	**Research**	**10500**	**1991**	**INF**
13	**Ringo**	**Sales**	**15500**	**1988**	**INF**

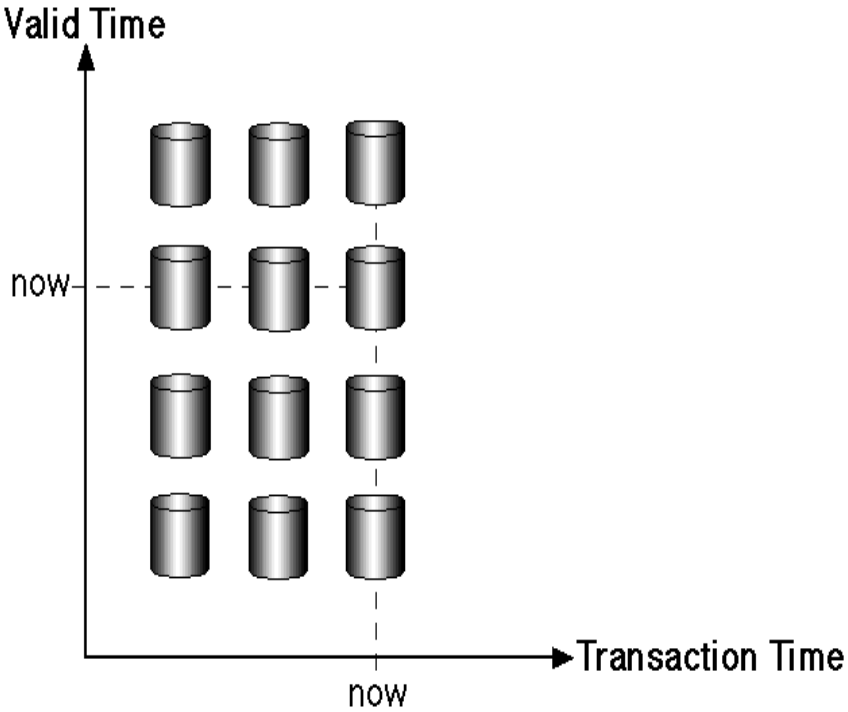
The above valid-time table stores the history of the employees with respect to the real world. The attributes**ValidTimeStart** and **ValidTimeEnd** actually represent a time interval which is closed at its lower and open at its upper bound. Thus, we see that during the time period [1985 - 1990), employee John was working in the research department, having a salary of 11000. Then he changed to the sales department, still earning 11000. In 1993, he got a salary raise to 12000. The upper bound INF denotes that the tuple is valid until further notice. Note that it is now possible to store information about past states. We see that Paul was employed from 1988 until 1995. In the corresponding non-temporal table, this information was (physically) deleted when Paul left the company.

Different Forms of Temporal Databases

The two different notions of time - valid time and transaction time - allow the distinction of different forms of temporal databases. A **historical database** stores data with respect to valid time, a **rollback database** stores data with respect to transaction time. A **bitemporal database** stores data with respect to both valid time and transaction time.

As we mentioned above, commercial DBMS are said to store only a single state of the real world, usually the most recent state. Such databases usually are called **snapshot databases**. A snapshot database in the context of valid time and transaction time is depicted in the following picture:





On the other hand, a **bitemporal DBMS** such as TimeDB stores the history of data with respect to both valid time and transaction time. Note that the history of when data was stored in the database (transaction time) is limited to past and present database states, since it is managed by the system directly which does not know anything about future states.

A table in the bitemporal relational DBMS TimeDB may either be a snapshot table (storing only current data), a valid-time table (storing when the data is valid wrt. the real world), a transaction-time table (storing when the data was recorded in the database) or a bitemporal table (storing both valid time and transaction time). An extended version of SQL allows to specify which kind of table is needed when the table is created. Existing tables may also be altered (schema versioning). Additionally, it supports **temporal queries**, **temporal modification statements** and**temporal constraints**.

The states stored in a bitemporal database are sketched in the picture below. Of course, a temporal DBMS

such as TimeDB does not store each database state separately as depicted in the picture below. It stores valid time and/or transaction time for each tuple, as described above.

3.5 Spatial database concepts and architecture

A **spatial database** is a database that is optimized to store and query data that represents objects defined in a geometric space. Most spatial databases allow representing simple geometric objects such as points, lines and polygons. Some spatial databases handle more complex structures such as 3D objects, topological coverages, linear networks, and TINs. While typical databases are designed to manage various numeric and character types of data, additional functionality needs to be added for databases to process spatial data types efficiently. These are typically called *geometry* or *feature*. The Open Geospatial Consortium created the Simple Features specification and sets standards for adding spatial functionality to database systems.

Features

Database systems use indexes to quickly look up values and the way that most databases index data is not optimal for spatial queries. Instead, spatial databases use a spatial index to speed up database operations.

In addition to typical SQL queries such as SELECT statements, spatial databases can perform a wide variety of spatial operations. The following operations and many more are specified by the Open Geospatial Consortium standard:

- Spatial Measurements: Computes line length, polygon area, the distance between geometries, etc.
- Spatial Functions: Modify existing features to create new ones, for example by providing a buffer around them, intersecting features, etc.
- Spatial Predicates: Allows true/false queries about spatial relationships between geometries. Examples include "do two polygons overlap" or 'is there a residence located within a mile of the area we are planning to build the landfill?' (see DE-9IM)

•Geometry Constructors: Creates new geometries, usually by specifying the vertices (points or nodes) which define the shape.

•Observer Functions: Queries which return specific information about a feature such as the location of the center of a circle

Some databases support only simplified or modified sets of these operations, especially in cases of NoSQL systems like MongoDB and CouchDB.

3.6 Deductive databases and Query processing

A **Deductive database** is a database system that can make deductions (i.e., conclude additional facts) based on rules and facts stored in the (deductive) database.Datalog is the language typically used to specify facts, rules and queries in deductive databases. Deductive databases have grown out of the desire to combine logic programming with relational databases to construct systems that support a powerful formalism and are still fast and able to deal with very large datasets. Deductive databases are more expressive than relational databases but less expressive than logic programming systems. In recent years, deductive databases such as Datalog have found new application in data integration, information extraction, networking, program analysis, security, and cloud computing.

Deductive databases and logic programming

Deductive databases reuse a large number of concepts from logic programming; rules and facts specified in the deductive database language Datalog look very similar to those in Prolog. However, there are a number of important differences between deductive databases and logic programming:

•Order sensitivity and procedurality: In Prolog, program execution depends on the order of rules in the program and on the order of parts of rules; these properties are used by programmers to build efficient programs. In database languages (like SQL or Datalog), however, program execution is independent of the order of rules and facts.

•Special predicates: In Prolog, programmers can directly influence the procedural evaluation of the program with special predicates such as the cut, this has no correspondence in deductive databases.

- •Function symbols: Logic Programming languages allow function symbols to build up complex symbols. This is not allowed in deductive databases.
- •Tuple-oriented processing: Deductive databases use set-oriented processing while logic programming languages concentrate on one tuple at a time.

By staying with the simple data types, but adding recursion to this database language, one gets a language called (positive?) Datalog, which is the language underlying deductive databases. Deductive databases are an extension of relational databases which support more complex data modeling. In this section we will see how simple examples of deductive databases can be represented in Prolog, and we will see more of the limitations of Prolog.
A standard example in Prolog is a geneology database. An extensional relation stores
the parent relation: parent(X,Y) succeeds with X and Y if X has parent Y. (Maybe do an example consisting of some English monarchs?) Given this parent relation, we can define the ancestor relation as follows:

```
ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

This says that X has ancestor Y if X has parent Y; and X has ancestor Y if there is a Z such that X has parent Z and Z has ancestor Y. Given a definition for the parent relation as follows:

```
parent(elizabeth_II, charles_??).

etc.
```

we can query about ancestors, such as:

```
:- ancestor(elizabeth_II,X).
```

and find all of Queen Elizabeth's ancestors.

3.7 Mobile Databases

A **mobile database** is either a stationary database that can be connected to by a mobile computing device - such as smart phones or PDAs - over a mobile network, or a database which is actually carried by the mobile device. This could be a list of contacts, price information, distance travelled, or any other information.

Many applications require the ability to download information from an information repository and operate on this information even when out of range or disconnected. An example of this is a mobile workforce. In this scenario, a user would require access to update information from files in the home directories on a server or customer records from a database. This type of access and work load generated by such users is different from the traditional workloads seen in client–server systems of today.

Mobile databases are highly concentrated in the retail and logistics industries. They are increasingly being used in aviation and transportation industry.

Considerations

- Mobile users must be able to work without a network connection due to poor or even non-existent connections. A cache could maintained to hold recently accessed data and transactions so that they are not lost due to connection failure. Users might not require access to truly live data, only recently modified data, and uploading of changing might be deferred until reconnected.
- Bandwidth must be conserved (a common requirement on wireless networks that charge per megabyte or data transferred).
- Mobile computing devices tend to have slower CPUs and limited battery life.
- Users with multiple devices (e.g. smartphone and tablet) may need to synchronize their devices to a centralized data store. This may require application-specific automation features.
- Users may change location geographically and on the network. Usually dealing with this is left to the operating system, which is responsible for maintaining the wireless network connection.

IBM mobile Database

IBM® Mobile Database is a new no-charge download for Android mobile devices. It offers a tight integration

between a mobile environment and an existing IBM DB2 or IBM Informix database. IBM Mobile Database makes it easier for mobile developers to develop and assemble applications for Android devices. Together with IBM solidDB, IBM Mobile Database provides the capability to synchronize data with DB2 and Informix databases. With IBM Mobile Database, you can use your DB2 or Informix data applications offline on your Android device and then synchronize the database later, when you are online.

3.8 Geographic Information Systems

A **geographic information system** (**GIS**) is a system designed to capture, store, manipulate, analyze, manage, and present all types of geographical data. The acronym*GIS* is sometimes used for **geographical information science** or **geospatial information studies** to refer to the academic discipline or career of working with geographicinformation systems and is a large domain within the broader academic discipline of Geoinformatics. In the simplest terms, GIS is the merging of cartography, statistical analysis, and computer science technology.

A GIS can be thought of as a system—it digitally creates and "manipulates" spatial areas that may be jurisdictional, purpose, or application-oriented. Generally, a GIS is custom-designed for an organization. Hence, a GIS developed for an application, jurisdiction, enterprise, or purpose may not be necessarily interoperable or compatible with a GIS that has been developed for some other application, jurisdiction, enterprise, or purpose. What goes beyond a GIS is a spatial data infrastructure, a concept that has no such restrictive boundaries.

In a general sense, the term describes any information system that integrates, stores, edits, analyzes, shares, and displays geographic information for informing decision making. GIS applications are tools that allow users to create interactive queries (user-created searches), analyze spatial information, edit data in maps, and present the results of all these operations.[2] Geographic information science is the science underlying geographic concepts, applications, and systems.

The first known use of the term "Geographic Information System" was by Roger Tomlinson in the year 1968 in his paper "A Geographic Information System for Regional Planning". Tomlinson is also acknowledged as the "father of GIS".

Data and GIS

Data in many different forms can be entered into GIS. Data that are already in map form can be included in GIS. This includes such information as the location of rivers and roads, hills and valleys. Digital, or computerized, data can also be entered into GIS. An example of this kind of information is data collected by satellites that show land use—the location of farms, towns, or forests. GIS can also include data in table form, such as population information. GIS technology allows all these different types of information, no matter their source or original format, to be overlaid on top of one another on a single map.

Putting information into GIS is called data capture. Data that are already in digital form, such as images taken by satellites and most tables, can simply be uploaded into GIS. Maps must be scanned, or converted into digital information.

GIS must make the information from all the various maps and sources align, so they fit together. One reason this is necessary is because maps have different scales. A scale is the relationship between the distance on a map and the actual distance on Earth. GIS combines the information from different sources in such a way that it all has the same scale.

GIS Maps

Once all of the desired data have been entered into a GIS system, they can be combined to produce a wide variety of individual maps, depending on which data layers are included. For instance, using GIS technology, many kinds of information can be shown about a single city. Maps can be produced that relate such information as average income, book sales, and voting patterns. Any GIS data layer can be added or subtracted to the same map.

GIS maps can be used to show information about number and density. For example, GIS can be used to show how many doctors there are in different areas compared with the population. They can also show what is near what, such as which homes and businesses are in areas prone to flooding.

Unit 4: New database applications and environments

4.1 Data Warehousing and Data Mining

Data, Information, and Knowledge

Data

Data are any facts, numbers, or text that can be processed by a computer. Today, organizations are accumulating vast and growing amounts of data in different formats and different databases. This includes:

- operational or transactional data such as, sales, cost, inventory, payroll, and accounting
- nonoperational data, such as industry sales, forecast data, and macro economic data
- meta data - data about the data itself, such as logical database design or data dictionary definitions

Information

The patterns, associations, or relationships among all this data can provide information. For example, analysis of retail point of sale transaction data can yield information on which products are selling and when.

Knowledge

Information can be converted into knowledge about historical patterns and future trends. For example, summary information on retail supermarket sales can be analyzed in light of promotional efforts to provide knowledge of consumer buying behavior. Thus, a manufacturer or retailer could determine which items are most susceptible to promotional efforts.

What is a Data Warehouse?

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources. This helps in:

- Maintaining historical records
- Analyzing the data to gain a better understanding of the business and to improve the business.

In addition to a relational database, a data warehouse environment can include an extraction, transportation, transformation, and loading (ETL) solution, statistical analysis, reporting, data mining capabilities, client analysis tools, and other applications that manage the process of gathering data, transforming it into useful, actionable information, and delivering it to business users.

Characteristics

Subject Oriented

Data warehouses are designed to help you analyze data. For example, to learn more about your company's sales data, you can build a data warehouse that concentrates on sales. Using this data warehouse, you can answer questions such as "Who was our best customer for this item last year?" or "Who is likely to be our best customer next year?" This ability to define a data warehouse by subject matter, sales in this case, makes the data warehouse subject oriented.

Integrated

Integration is closely related to subject orientation. Data warehouses must put data from disparate sources into a consistent format. They must resolve such problems as naming conflicts and inconsistencies among units of measure. When they achieve this, they are said to be integrated.

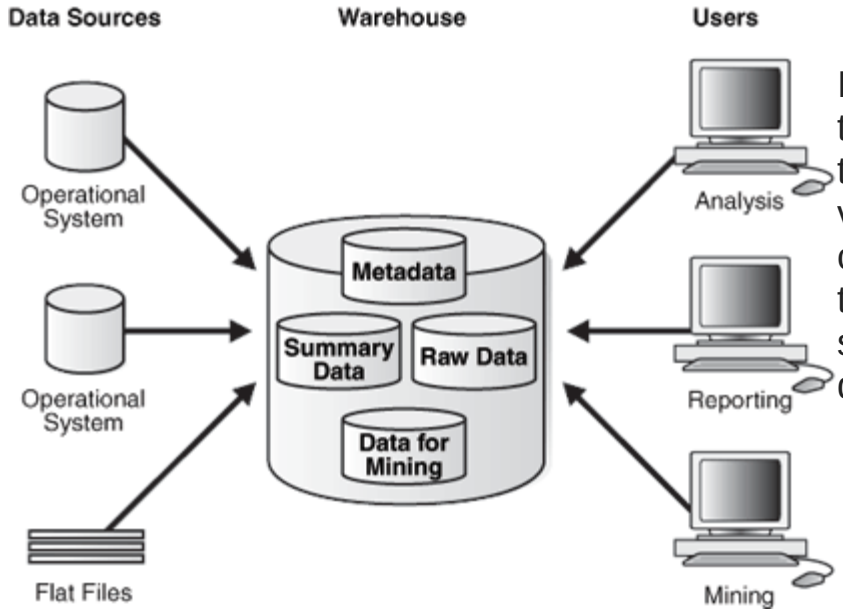
Nonvolatile

Nonvolatile means that, once entered into the data warehouse, data should not change. This is logical because the purpose of a data warehouse is to enable you to analyze what has occurred.

Time Variant

A data warehouse's focus on change over time is what is meant by the term time variant. In order to discover trends and identify hidden patterns and relationships in business, analysts need large amounts of data. This is very much in contrast to **online transaction processing (OLTP)** systems, where performance requirements demand that historical data be moved to an archive.

Architecture



In Figure 1-2, the metadata and raw data of a traditional OLTP system is present, as is an additional type of data, summary data. Summaries are very valuable in data warehouses because they pre-compute long operations in advance. For example, a typical data warehouse query is to retrieve something such as August sales. A summary in an Oracle database is called a **materialized view**.

Extracting Information from a Data Warehouse

You can extract information from the masses of data stored in a data warehouse by analyzing the data. The Oracle Database provides several ways to analyze data:

•A wide array of statistical functions, including descriptive statistics, hypothesis testing, correlations analysis, test for distribution fit, cross tabs with Chi-square statistics, and analysis of variance (ANOVA); these functions are described in the *Oracle Database SQL Language Reference*.

•OLAP

•Data Mining

Data Mining

Data mining uses large quantities of data to create models. These models can provide insights that are revealing, significant, and valuable. For example, data mining can be used to:

- Predict those customers likely to change service providers.
- Discover the factors involved with a disease.
- Identify fraudulent behavior.

Data mining is not restricted to solving business problems. For example, data mining can be used in the life sciences to discover gene and protein targets and to identify leads for new drugs.

Oracle Data Mining performs data mining in the Oracle Database. Oracle Data Mining does not require data movement between the database and an external mining server, thereby eliminating redundancy, improving efficient data storage and processing, ensuring that up-to-date data is used, and maintaining data security.

Oracle Data Mining Functionality

Oracle Data Mining supports the major data mining functions. There is at least one algorithm for each data mining function.

Oracle Data Mining supports the following data mining functions:

•Classification: Grouping items into discrete classes and predicting which class an item belongs to;

classification algorithms are Decision Tree, Naive Bayes, Generalized Linear Models (Binary Logistic Regression), and Support Vector Machines.

•Regression: Approximating and predicting continuous numerical values; the algorithms for regression are Support Vector Machines and Generalized Linear Models (Multivariate Linear Regression).

•Anomaly Detection: Detecting anomalous cases, such as fraud and intrusions; the algorithm for anomaly detection is one-class Support Vector Machines.

•Attribute Importance: Identifying the attributes that have the strongest relationships with the target attribute (for example, customers likely to churn); the algorithm for attribute importance is Minimum Descriptor Length.

•Clustering: Finding natural groupings in the data that are often used for identifying customer segments; the algorithms for clustering are *k*-Means and O-Cluster.

•Associations: Analyzing "market baskets", items that are likely to be purchased together; the algorithm for associations is a priori.

•Feature Extraction: Creating new attributes (features) as a combination of the original attributes; the algorithm for feature extraction is Non-Negative Matrix Factorization.

In addition to mining structured data, ODM permits mining of text data (such as police reports, customer comments, or physician's notes) or spatial data.

4.2 Multimedia

A multimedia database is a database that hosts one or more primary media file types such as .txt (documents), .jpg (images), .swf (videos), .mp3 (audio), etc. And loosely fall into three main categories:

- •Static media (time-independent, i.e. images and handwriting)
- •Dynamic media (time-dependent, i.e. video and sound bytes)
- •Dimensional media (i.e. 3D games or computer-aided drafting programs- CAD)

All primary media files are stored in binary strings of zeros and ones, and are encoded according to file

type.

The term "data" is typically referenced from the computer point of view, whereas the term "multimedia" is referenced from the user point of view.

Types of Multimedia Databases

There are numerous different types of multimedia databases, including:

- The Authentication **Multimedia** Database (also known as a Verification Multimedia Database, i.e. retina scanning), is a 1:1 data comparison
- The Identification Multimedia Database is a data comparison of one-to-many (i.e. passwords and personal identification numbers
- A newly-emerging type of multimedia database, is the **Biometrics** Multimedia Database; which specializes in automatic human verification based on the algorithms of their behavioral or physiological profile.

This method of identification is superior to traditional multimedia database methods requiring the typical input of personal identification numbers and passwords-

Due to the fact that the person being identified does not need to be physically present, where the identification check is taking place.

This removes the need for the person being scanned to remember a PIN or password. Fingerprint identification technology is also based on this type of multimedia database.

Difficulties Involved with Multimedia Databases

The difficulty of making these different types of multimedia databases readily accessible to humans is:

- The tremendous amount of bandwidth they consume;
- Creating Globally-accepted data-handling platforms, such as Joomla, and the special considerations that these new multimedia database structures require.
- Creating a Globally-accepted operating system, including applicable storage and resource management programs need to accommodate the vast Global multimedia information hunger.

•Multimedia databases need to take into accommodate various human interfaces to handle 3D-interactive objects, in an logically-perceived manner (i.e. SecondLife.com).
•Accommodating the vast resources required to utilize artificial intelligence to it's fullest potential- including computer sight and sound analysis methods.
•The historic relational databases (i.e the **Binary** Large Objects – BLOBs- developed for SQL databases to store multimedia data) do not conveniently support content-based searches for multimedia content.

Oracle Multimedia

Oracle Multimedia is a feature that enables Oracle Database to store, manage, and retrieve multimedia data in an integrated manner with other enterprise information. With support for images, medical images, audio, and video data, it provides a high performance, highly available, robust, scalable, and secure platform for a wide range of multimedia intensive applications. These include medical image management in healthcare and life sciences; multimedia asset management for museums, corporations, and the public sector; check image management at financial institutions; asset management for web publishing, video security, and many more.

Oracle Multimedia includes comprehensive support for the Digital Image and Communications in Medicine (DICOM) standard, widely adopted for medical images, videos, and structured reports.

4.3 Mobility {

Together with wireless communication technologies portable computers and personal digital assistants (PDA) provide a pervasive base for mobile computing. As the mobile computing technology matures, millions of people will become mobile users communicating with one another and accessing various information resources. Mobile computing involves mobility of users, hardware, software and data.

}

4.4 Multi databases

4.5 Native XML databases (NXD)

An **XML database** is a data persistence software system that allows data to be stored in XML format. These data can then be queried, exported and serialized into the desired format. XML databases are usually associated with document-oriented databases.

Two major classes of XML database exist:[1]

1.XML-enabled: these may either map XML to traditional database structures (such as a relational database), accepting XML as input and rendering XML as output, or more recently support native XML types within the traditional database. This term implies that the database processes the XML itself (as opposed to relying on middleware).
2.**Native XML** (NXD): the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files.

Native XML databases

The term "native XML database" (NXD) can lead to confusion. Many NXDs do not function as standalone databases at all, and do not really store the native (text) form.

The formal definition from the XML:DB initiative (which appears to be inactive since 2003) states that a native XML database:

- Defines a (logical) model for an XML document — as opposed to the data in that document — and stores and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Examples of such models include the XPath data model, the XML Infoset, and the models implied by the DOM and the events in SAX 1.0.
- Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
- Need not have any particular underlying physical storage model. For example, NXDs can use relational, hierarchical, or object-oriented database structures, or use a proprietary storage format (such

as indexed, compressed files).
Additionally, many XML databases provide a logical model of grouping documents, called "collections". Databases can set up and manage many collections at one time. In some implementations, a hierarchy of collections can exist, much in the same way that an operating system's directory-structure works.

All XML databases now support at least one form of querying syntax. Minimally, just about all of them support XPath for performing queries against documents or collections of documents. XPath provides a simple pathing system that allows users to identify nodes that match a particular set of criteria.

In addition to XPath, many XML databases support XSLT as a method of transforming documents or query-results retrieved from the database. XSLT provides a declarative language written using an XML grammar. It aims to define a set of XPath filters that can transform documents (in part or in whole) into other formats including plain text, XML, or HTML.

Many XML databases also support XQuery to perform querying. XQuery includes XPath as a node-selection method, but extends XPath to provide transformational capabilities. Users sometimes refer to its syntax as "FLWOR" (pronounced 'Flower') because the query may include the following clauses: 'for', 'let', 'where', 'order by' and 'return'. Traditional RDBMS vendors (who traditionally had SQL-only engines), are now shipping with hybrid SQL and XQuery engines. Hybrid SQL/XQuery engines help to query XML data alongside the relational data, in the same query expression. This approach helps in combining relational and XML data.

Most XML Databases support a common vendor neutral API called the XQuery API for Java (XQJ). The XQJ API was developed at the JCP as a standard interface to an XML/XQuery data source, enabling a Java developer to submit queries conforming to the World Wide Web Consortium (W3C) XQuery 1.0 specification and to process the results of such queries. Ultimately the XQJ API is to XML Databases and XQuery as the JDBC API is to Relational Databases and SQL.

4.6 Internet

An **online database** is a database accessible from a network, including from the Internet.

It differs from a local database, held in an individual computer or its attached storage, such as a CD.

- For the system or software designed to manage a database, see Database management systems (DBMS)
- For information on the structure of a database itself, see Database
- For information on the programs for searching a database, see Search engines

Currently, there are several database products designed specifically as hosted databases delivered as software as a service products. These differ from typical traditional databases such as Oracle, Microsoft SQL Server, Sybase, etc. Some of the differences are:

- These online databases are delivered primarily via a web browser
- They are often purchased by a monthly subscription
- They embed common collaboration features such as sharing, email notifications, etc.

Internet Movie Database (**IMDb**) is an online database of information related to films, television programs and video games. This includes actors, production crew personnel, and fictional characters featured in these three visual entertainment media. Since 2008, a feature also enables U.S. users to instantly view over 6,000 movies and television shows from CBS, Sony and various independent film makers.

Cloud Database

A **cloud database** is a database that typically runs on a cloud computing platform, such as Amazon EC2, GoGrid and Rackspace. There are two common deployment models: users can run databases on the cloud independently, using a virtual machine image, or they can purchase access to a database service, maintained by a cloud database provider. Of the databases available on the cloud, some are SQL-based and some use a NoSQL data model.

There are two primary methods to run a database on the cloud:

- **Virtual machine Image** - cloud platforms allow users to purchase virtual machine instances for a limited time. It is possible to run a database on these virtual machines. Users can either upload their own machine image with a database installed on it, or use ready-made machine images that already include an optimized installation of a database. For example, Oracle provides a ready-made machine image with

an installation of Oracle Database 11g Enterprise Edition on Amazon EC2.

- **Database as a service** - some cloud platforms offer options for using a database as a service, without physically launching a virtual machine instance for the database. In this configuration, application owners do not have to install and maintain the database on their own. Instead, the database service provider takes responsibility for installing and maintaining the database, and application owners pay according to their usage. For example, Amazon Web Services provides three database services as part of its cloud offering, SimpleDB, a NoSQL key-value store, Amazon Relational Database Service, an SQL-based database service with a MySQL interface, and DynamoDB.

A third option is managed database hosting on the cloud, where the database is not offered as a service, but the cloud provider hosts the database and manages it on the application owner's behalf. For example, cloud provider Rackspace offers managed hosting for MySQL databases.

Data model

It is also important to differentiate between cloud databases which are relational as opposed to non-relational or NoSQL:

- **SQL database**, such as NuoDB, Oracle Database, Microsoft SQL Server, and MySQL, are one type of database which can be run on the cloud (either as a Virtual Machine Image or as a service, depending on the vendor). SQL databases are difficult to scale, meaning they are not natively suited to a cloud environment, although cloud database services based on SQL are attempting to address this challenge.
- **NoSQL databases**, such as Apache Cassandra, CouchDB and MongoDB, are another type of database which can run on the cloud. NoSQL databases are built to service heavy read/write loads and are able scale up and down easily, and therefore they are more natively suited to running on the cloud. However, most contemporary applications are built around an SQL data model, so working with NoSQL databases often requires a complete rewrite of application code.

Unit 5: Database Related Standards

5.1 SQL standards

SQL was adopted as a standard by the American National Standards Institute (ANSI) in 1986 as SQL-86 and the International Organization for Standardization (ISO) in 1987. Nowadays the standard is subject to continuous improvement by the Joint Technical Committee *ISO/IEC JTC 1, Information technology, Subcommittee SC 32, Data management and interchange* which affiliate to ISO as well as IEC. It is commonly denoted by the pattern: *ISO/IEC 9075-n:yyyy Part n: title*, or, as a shortcut,*ISO/IEC 9075*.

ISO/IEC 9075 is complemented by *ISO/IEC 13249: SQL Multimedia and Application Packages* (SQL/MM) which defines SQL based interfaces and packages to widely spread applications like video, audio and spatial data.

Until 1996, the National Institute of Standards and Technology (NIST) data management standards program certified SQL DBMS compliance with the SQL standard. Vendors now self-certify the compliance of their products.

The original SQL standard declared that the official pronunciation for SQL is "es queue el".[10] Many English-speaking database professionals still use the original pronunciation /ˈsiːkwəl/ (like the word "sequel"), including Donald Chamberlin himself.

The SQL standard has gone through a number of revisions:

Year	Name	Alias	Comments
1986	SQL-86	SQL-87	First formalized by ANSI.
1989	SQL-89	FIPS127-1	Minor revision, in which the major addition were integrity constraints. Adopted as FIPS 127-1.

1992	SQL-92	SQL2, FIPS 127-2	Major revision (ISO 9075), *Entry Level* SQL-92 adopted as FIPS 127-2.
1999	SQL:1999	SQL3	Added regular expression matching, recursive queries (e.g. transitive closure), triggers, support for procedural and control-of-flow statements, non-scalar types, and some object-oriented features (e.g. structured types). Support for embedding SQL in Java (SQL/OLB) and vice-versa (SQL/JRT).
2003	SQL:2003	SQL 2003	Introduced XML-related features (SQL/XML), *window functions*, standardized sequences, and columns with auto-generated values (including identity-columns).
2006	SQL:2006	SQL 2006	ISO/IEC 9075-14:2006 defines ways in which SQL can be used in conjunction with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database and publishing both XML and conventional SQL-data in XML form. In addition, it enables applications to integrate into their SQL code the use of XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.
2008	SQL:2008	SQL 2008	Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers. Adds the TRUNCATE statement.
2011	SQL:2011		

Interested parties may purchase SQL standards documents from ISO, IEC or ANSI. A draft of SQL:2008 is freely available as a zip archive

5.2 SQL 1999

SQL:1999 (also called SQL 3) was the fourth revision of the SQL database query language. It introduced a large number of new features, many of which required clarifications in the subsequent SQL:2003. The latest revision of the standard is SQL:2011.

Summary

The ISO standard documents were published between 1999 and 2002 in several installments, the first one consisting of multiple parts. Unlike previous editions, the standard's name used a colon instead of a hyphen for consistency with the names of other ISO standards. The first installment of SQL:1999 had five parts:

- SQL/Framework ISO/IEC 9075-1:1999
- SQL/Foundation ISO/IEC 9075-2:1999
- SQL/CLI : an updated definition of the extension Call Level Interface, originally published in 1995, also known as CLI-95 ISO/IEC 9075-3:1999
- SQL/PSM : an updated definition of the extension Persistent Stored Modules, originally published in 1996, also known as PSM-96 ISO/IEC 9075-4:1999
- SQL/Bindings ISO/IEC 9075-5:1999

Three more parts, also considered part of SQL:1999 were published subsequently:

- SQL/MED Management of External Data (SQL:1999 part 9) ISO/IEC 9075-9:2001
- SQL/OLB Object Language Bindings (SQL:1999 part 10) ISO/IEC 9075-10:2000
- SQL/JRT SQL Routines and Types using the Java Programming Language (SQL:1999 part 13) ISO/IEC 9075-13:2002

New features

Data types

Boolean data types

The SQL:1999 standard calls for a Boolean type, but many commercial SQL Servers (Microsoft SQL Server 2005, Oracle 9i, IBM DB2) do not support it as a column type, variable type or allow it in the results

set. MySQL interprets "BOOLEAN" as a synonym for TINYINT (8-bit signed integer).

Distinct user-defined types of power
Sometimes called just *distinct types*, these were introduced as an optional feature (S011) to allow existing atomic types to be extended with a distinctive meaning to create a new type and thereby enabling the type checking mechanism to detect some logical errors, e.g. accidentally adding an age to a salary. For example:

```
CREATE TYPE age AS INTEGER FINAL;
CREATE TYPE salary AS INTEGER FINAL;
```

creates two different and incompatible types. The SQL distinct types use name equivalence not structural equivalence like typedefs in C. It's still possible to perform compatible operations on (columns or data) of distinct types by using an explicit type CAST.

Few SQL systems support these. IBM DB2 is one those supporting them. Oracle database does not currently support them, recommending instead to emulate them by a one-place structured type.

Structured user-defined types
These are the backbone of the object-relational database extension in SQL:1999. They are analogous to classes in objected-oriented programming languages. SQL:1999 allows only single inheritance.

Common table expressions and recursive queries
SQL:1999 added a WITH [RECURSIVE] construct allowing recursive queries, like transitive closure, to be specified in the query language itself; see common table expressions.

Some OLAP capabilities
GROUP BY was extended with ROLLUP, CUBE, and GROUPING SETS.

Role-based access control
Full support for RBAC via CREATE ROLE.

5.3 SQL:2003

SQL:2003 is the fifth revision of the SQL database query language. The latest revision of the standard is SQL:2011.

Summary

The SQL:2003 standard makes minor modifications to all parts of SQL:1999 (also known as SQL3), and officially introduces a few new features such as:

- XML-related features (SQL/XML)
- Window functions
- the sequence generator, which allows standardized sequences
- two new column types: auto-generated values and identity-columns
- the new MERGE statement
- extensions to the CREATE TABLE statement, to allow "CREATE TABLE AS" and "CREATE TABLE LIKE"
- removal of the poorly implemented "BIT" and "BIT VARYING" data types
- OLAP capabilities (initially added in SQL:1999) were extended with a window function.

Documentation availability

The SQL standard is not freely available. SQL:2003 may be purchased from ISO or ANSI. A late draft is available as a zip archive from Whitemarsh Information Systems Corporation. The zip archive contains a number of PDF files that define the parts of the SQL:2003 specification.

- ISO/IEC 9075(1-4,9-11,13,14):2003 CD-ROM (352 CHF, or approximately 225 EUR, to order the CD)
- ISO/IEC 9075-1:2003 – Framework (SQL/Framework)
- ISO/IEC 9075-2:2003 – Foundation (SQL/Foundation)
- ISO/IEC 9075-3:2003 – Call-Level Interface (SQL/CLI)
- ISO/IEC 9075-4:2003 – Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9:2003 – Management of External Data (SQL/MED)
- ISO/IEC 9075-10:2003 – Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11:2003 – Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13:2003 – SQL Routines and Types Using the Java Programming Language (SQL/JRT)
- ISO/IEC 9075-14:2003 – XML-Related Specifications (SQL/XML)

5.4 Object Data Management Group (ODMG) version 3.0 standard

The **Object Data Management Group** (**ODMG**) was conceived in the summer of 1991 at a breakfast with object database vendors that was organized by Rick Cattell ofSun Microsystems. In 1998, the ODMG changed its name from the Object Database Management Group to reflect the expansion of its efforts to include specifications for both object database and object-relational mapping products.

The primary goal of the ODMG was to put forward a set of specifications that allowed a developer to write portable applications for object database and object-relational mapping products. In order to do that, the data schema, programming language bindings, and data manipulation and query languages needed to be portable.

Between 1993 and 2001, the ODMG published five revisions to its specification. The last revision was ODMG version 3.0, after which the group disbanded.

Major components of the ODMG 3.0 specification

- *Object Model.* This was based on the Object Management Group's Object Model. The OMG core model was designed to be a common denominator for object request brokers, object database systems, object programming languages, etc. The ODMG designed a profile by adding components to the OMG core object model.
- *Object Specification Languages.* The ODMG Object Definition Language (ODL) was used to define the object types that conform to the ODMG Object Model. The ODMG Object Interchange Format (OIF) was used to dump and load the current state to or from a file or set of files.
- *Object Query Language (OQL).* The ODMG OQL was a declarative (nonprocedural) language for query and updating. It used SQL as a basis, where possible, though OQL supports more powerful object-oriented capabilities.
- *C++ Language Binding.* This defined a C++ binding of the ODMG ODL and a C++ Object Manipulation Language (OML). The C++ ODL was expressed as a library that provides classes and functions to implement the concepts defined in the ODMG Object Model. The C++ OML syntax and semantics are those of standard C++ in the context of the standard class library. The C++ binding also provided a mechanism to invoke OQL.
- *Smalltalk Language Binding.* This defined the mapping between the ODMG ODL and Smalltalk, which was based on the OMG Smalltalk binding for the OMG Interface Definition Language (IDL). The Smalltalk

binding also provided a mechanism to invoke OQL.

•*Java Language Binding.* This defined the binding between the ODMG ODL and the Java programming language as defined by the Java 2 Platform. The Java binding also provided a mechanism to invoke OQL.

Status

ODMG 3.0 was published in book form in 2000.[1] By 2001, most of the major object database and object-relational mapping vendors claimed conformance to the ODMG Java Language Binding. Compliance to the other components of the specification was mixed. In 2001, the ODMG Java Language Binding was submitted to the Java Community Process as a basis for the Java Data Objects specification. The ODMG member companies then decided to concentrate their efforts on the Java Data Objects specification. As a result, the ODMG disbanded in 2001.

In 2004, the Object Management Group (OMG) was granted the right to revise the ODMG 3.0 specification as an OMG specification by the copyright holder, Morgan Kaufmann Publishers. In February 2006, the OMG announced the formation of the Object Database Technology Working Group (ODBT WG) and plans to work on the**4th generation of an object database standard**.

5.5 Standards for interoperability and integration e.g. Web Services, SOAP

Interoperability is the ability of diverse systems and organizations to work together (inter-operate). While the term was initially defined for information technology or systems engineering services to allow for information exchange, a more broad definition takes into account social, political, and organizational factors that impact system to system performance.

Data integration involves combining data residing in different sources and providing users with a unified view of these data. This process becomes significant in a variety of situations, which include both commercial (when two similar companies need to merge their databases) and scientific (combining research results from different bioinformatics repositories, for example) domains. Data integration appears with increasing frequency as the volume and the need to share existing data explodes. It has become the focus of extensive theoretical work, and numerous open problems remain unsolved. In management circles,

people frequently refer to data integration as "Enterprise Information Integration" (EII).

Web Services

A **web service** is a method of communication between two electronic devices over the World Wide Web. A **web service** is a software function provided at a network address over the web or the cloud; it is a service that is "always on" as in the concept of utility computing.

The W3C defines a "Web service" as:

[...] a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

The W3C also states:

We can identify two major classes of Web services:

REST-compliant Web services, in which the primary purpose of the service is to manipulate XML representations of Web resources using a uniform set of "stateless" operations; and

arbitrary Web services, in which the service may expose an arbitrary set of operations.

Web API

A web API is a development in web services where emphasis has been moving to simpler representational state transfer (REST) based communications. RESTful APIs do not require XML-based web service protocols (SOAP and WSDL) to support their light-weight interfaces.
Web APIs allow the combination of multiple web resources into new applications known as mashups.

WSDL version 2.0 offers support for binding to all the HTTP request methods (beyond GET and POST as in version 1.1) allowing for a "RESTful web services" approach.

XML web services

XML web services use Extensible Markup Language (XML) messages that follow the SOAP standard and have been popular with the traditional enterprises. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language(WSDL). The latter is not a requirement of a SOAP *endpoint*, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Apache Axis2, Apache CXF, Spring, gSOAP being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

Automated design methods

Automated tools can aid in the creation of a web service. For services using WSDL it is possible to either automatically generate WSDL for existing classes (a bottom-up strategy) or to generate a class skeleton given existing WSDL (a top-down strategy).

- A developer using a bottom up method writes implementing classes first (in some programming language), and then uses a WSDL generating tool to expose methods from these classes as a web service. This is simpler to develop but may be harder to maintain if the original classes are subject to frequent change.
- A developer using a top down method writes the WSDL document first and then uses a code generating tool to produce the class skeleton, to be completed as necessary. This way is generally considered more difficult but can produce cleaner designs and is generally more resistant to change. As long as the message formats between sender and receiver do not change, changes in the sender and receiver themselves do not affect the web service. The technique is also referred to as "contract first" since the WSDL (or contract between sender and receiver) is the starting point.

SOAP

SOAP, originally defined as **Simple Object Access Protocol**, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on XML Information Set for its message format, and usually relies on other Application Layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

SOAP was designed as an object-access protocol in 1998 by Dave Winer, Don Box, Bob Atkinson, and Mohsen Al-Ghosein for Microsoft, where Atkinson and Al-Ghosein were working at the time.[1] The SOAP specification is currently maintained by the XML Protocol Working Group of the World Wide Web Consortium.

SOAP originally stood for 'Simple Object Access Protocol' but this acronym was dropped with Version 1.2 of the standard.Ver sion 1.2 became a W3C recommendation on June 24, 2003.

After SOAP was first introduced, it became the underlying layer of a more complex set of Web Services, based on Web Services Description Language (WSDL) and Universal Description Discovery and Integration (UDDI). These services, especially UDDI, have proved to be of far less interest, but an appreciation of them gives a more complete understanding of the expected role of SOAP compared to how web services have actually evolved.

Specification

The SOAP specification defines the messaging framework which consists of:

- The **SOAP processing model** defining the rules for processing a SOAP message
- The **SOAP extensibility model** defining the concepts of SOAP features and SOAP modules
- The **SOAP underlying protocol binding** framework describing the rules for defining a binding to an underlying protocol that can be used for exchanging SOAP messages between SOAP nodes
- The **SOAP message construct** defining the structure of a SOAP message

Processing model
The SOAP processing model describes a distributed processing model, its participants, the **SOAP nodes**, and how a SOAP receiver processes a SOAP message. The following SOAP nodes are defined:

SOAP sender
A SOAP node that transmits a SOAP message.

SOAP receiver
A SOAP node that accepts a SOAP message.

SOAP message path
The set of SOAP nodes through which a single SOAP message passes.

Initial SOAP sender (Originator)
The SOAP sender that originates a SOAP message at the starting point of a SOAP message path.

SOAP intermediary
A SOAP intermediary is both a SOAP receiver and a SOAP sender and is targetable from within a SOAP message. It processes the SOAP header blocks targeted at it and acts to forward a SOAP message towards an ultimate SOAP receiver.

Ultimate SOAP receiver
The SOAP receiver that is a final destination of a SOAP message. It is responsible for processing the contents of the SOAP body and any SOAP header blocks targeted at it. In some circumstances, a SOAP message might not reach an ultimate SOAP receiver, for example because of a problem at a SOAP intermediary. An ultimate SOAP receiver cannot also be a SOAP intermediary for the same SOAP message.

SOAP Building Blocks

A SOAP message is an ordinary XML document containing the following elements:

Element	Description	Required?
Envelope	**Identifies the XML document as a SOAP message.**	**Yes**
Header	**Contains header information.**	**No**
Body	**Contains call and response information.**	**Yes**
Fault	**Provides information about errors that occurred while processing the message.**	**No**

Example message

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Technical critique

Advantages

- SOAP is versatile enough to allow for the use of different transport protocols. The standard stacks use HTTP as a transport protocol, but other protocols such as JMS, SMTP or Message queue are also usable.
- Since the SOAP model tunnels fine in the HTTP post/response model, it can tunnel easily over existing firewalls and proxies, without modifications to the SOAP protocol, and can use the existing infrastructure.

Disadvantages

- When using standard implementations and the default SOAP/HTTP binding, the XML info set is serialized as XML. Because of the verbose XML format, SOAP can be considerably slower than competing middleware technologies such as CORBA or ICE. This may not be an issue when only small messages are sent.[9] To improve performance for the special case of XML with embedded binary objects, the Message Transmission Optimization Mechanism was introduced.
- When relying on HTTP as a transport protocol and not using WS-Addressing or an ESB, the roles of the interacting parties are fixed. Only one party (the client) can use the services of the other. Developers must use polling instead of notification in these common cases.

5.6 XML related specifications, e.g. XQuery, XPath.

Extensible Markup Language (**XML**) is a markup language that defines a set of rules for encoding documents in a formatthat is both human-readable and machine-readable. It is defined in the XML 1.0 Specification produced by the W3C, and several other related specifications, all free open standards.

The design goals of XML emphasize simplicity, generality, and usability over the Internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web services.

Many application programming interfaces (APIs) have been developed to aid software developers with processing XML data, and several schema systems exist to aid in the definition of XML-based languages.

As of 2009, hundreds of document formats using XML syntax have been developed, including RSS, Atom, SOAP, andXHTML. XML-based formats have become the default for many office-productivity tools, including Microsoft Office (Office Open XML), OpenOffice.org and LibreOffice (OpenDocument), and Apple's iWork. XML has also been employed as the base language for communication protocols, such as XMPP.

Key terminology

The material in this section is based on the XML Specification. This is not an exhaustive list of all the constructs that appear in XML; it provides an introduction to the key constructs most often encountered in day-to-day use.

(Unicode) character

By definition, an XML document is a string of characters. Almost every legal Unicode character may appear in an XML document.

Processor and application

The *processor* analyzes the markup and passes structured information to an *application*. The specification places requirements on what an XML processor must do and not do, but the application is outside its scope. The processor (as the specification calls it) is often referred to colloquially as an *XML parser*.

Markup and content

The characters making up an XML document are divided into *markup* and *content*, which may be distinguished by the application of simple syntactic rules. Generally, strings that constitute markup either begin with the character < and end with a >, or they begin with the character & and end with a ;. Strings of characters that are not markup are content. However, in a CDATA section, the delimiters <![CDATA[and]]> are classified as markup, while the text between them is classified as content. In addition, whitespace before and after the outermost element is classified as markup.

Tag
A markup construct that begins with < and ends with >. Tags come in three flavors:

- *start-tags*; for example: <section>
- *end-tags*; for example: </section>
- *empty-element tags*; for example: <line-break />

Element
A logical document component which either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start- and end-tags, if any, are the element's *content*, and may contain markup, including other elements, which are called *child elements*. An example of an element is <Greeting>Hello, world.</Greeting> (see hello world). Another is <line-break />.

Attribute
A markup construct consisting of a name/value pair that exists within a start-tag or empty-element tag. In the example (below) the element *img* has two attributes, *src*and *alt*:

> **<img** src="madonna.jpg" alt='Foligno Madonna, by Raphael'**/>**
> Another example would be
>
> **<step** number="3"**>**Connect A to B.**</step>**
> where the name of the attribute is "number" and the value is "3".

XML declaration
XML documents may begin by declaring some information about themselves, as in the following example:

> **<?xml** version="1.0" encoding="UTF-8" **?>**

Xquery

XQuery is a query and functional programming language that is designed to query and transform collections of structured and unstructured data, usually in the form of XML, text and with vendor specific extensions other data formats (JSON, binaries, ...).

XQuery 3.0 is the current candidate recommendation version being developed by the XML Query working group of theW3C.

XQuery 1.0 was developed by the XML Query working group of the W3C. The work was closely coordinated with the development of XSLT 2.0 by the XSL Working Group; the two groups shared responsibility for XPath 2.0, which is a subset of XQuery 1.0. XQuery 1.0 became a W3C Recommendation on January 23, 2007.

"The mission of the XML Query project is to provide flexible query facilities to extract data from real and virtual documents on the World Wide Web, therefore finally providing the needed interaction between the Web world and the database world. Ultimately, collections of XML files will be accessed like databases".**Examples**

The sample XQuery code below lists the unique speakers in each act of Shakespeare's play Hamlet, encoded in hamlet.xml

```
<html><head/><body>
{
  for $act in doc("hamlet.xml")//ACT
  let $speakers := distinct-values($act//SPEAKER)
  return
    <div>
      <h1>{ string($act/TITLE) }</h1>
      <ul>
      {
        for $speaker in $speakers
        return <li>{ $speaker }</li>
      }
      </ul>
    </div>
}
</body></html>
```

All XQuery constructs for performing computations are expressions. There are no statements, even though some of the keywords appear to suggest statement-like behaviors. To execute a function, the expression within the body is evaluated and its value is returned. Thus to write a function to double an input value, one simply writes:

```
declare function local:doubler($x) { $x * 2 }
```

To write a full query saying 'Hello World', one writes the expression:

```
"Hello World"
```

This style is common in functional programming languages. However, unlike most functional programming languages, XQuery 1.0 doesn't support higher-order functions(they first appear in the drafts for XQuery 3.0).

Applications
Below are a few examples of how XQuery can be used:

1. Extracting information from a database for use in a web service.
2. Generating summary reports on data stored in an XML database.
3. Searching textual documents on the Web for relevant information and compiling the results.
4. Selecting and transforming XML data to XHTML to be published on the Web.
5. Pulling data from databases to be used for the application integration.
6. Splitting up an XML document that represents multiple transactions into multiple XML documents.

Xpath

XPath, the **XML Path Language**, is a query language for selecting nodes from an XML document. In addition, XPath may be used to compute values (e.g., strings, numbers, or Boolean values) from the content of an XML document. XPath was defined by the World Wide Web Consortium (W3C).

Syntax and semantics

The most important kind of expression in XPath is a *location path*. A location path consists of a sequence of*location steps*. Each location step has three components:

- an *axis*
- a *node test*
- zero or more *predicates*.

An XPath expression is evaluated with respect to a *context node*. An Axis Specifier such as 'child' or 'descendant' specifies the direction to navigate from the context node. The node test and the predicate are used to filter the nodes specified by the axis specifier: For example the node test 'A' requires that all nodes navigated to must have label 'A'. A predicate can be used to specify that the selected nodes have certain properties, which are specified by XPath expressions themselves.

The XPath syntax comes in two flavours: the *abbreviated syntax*, is more compact and allows XPaths to be written and read easily using intuitive and, in many cases, familiar characters and constructs. The *full syntax* is more verbose, but allows for more options to be specified, and is more descriptive if read carefully.

Abbreviated syntax

The compact notation allows many defaults and abbreviations for common cases. Given source XML containing at least

```
<A>
 <B>
  <C/>
 </B>
</A>
```

the simplest XPath takes a form such as

- /A/B/C

which selects C elements that are children of B elements that are children of the A element that forms the outermost element of the XML document. The XPath syntax is designed to mimic URI (Uniform Resource Identifier) and Unix-style file path syntax.

More complex expressions can be constructed by specifying an axis other than the default 'child' axis, a node test other than a simple name, or predicates, which can be written in square brackets after any step. For example, the expression

- A//B/*[1]

selects the first element ('[1]'), whatever its name ('*'), that is a child ('/') of a B element that itself is a child or other, deeper descendant ('//') of an A element that is a child of the current context node (the expression does not begin with a '/'). If there are several suitable B elements in the document, this actually returns a set of all their first children. ("(A//B/*)[1]" returns just the first such node.)

Expanded syntax

In the full, unabbreviated syntax, the two examples above would be written

- /child::A/child::B/child::C
- child::A/descendant-or-self::node()/child::B/child::*[position()=1]

Here, in each step of the XPath, the **axis** (e.g. child or descendant-or-self) is explicitly specified, followed by :: and then the **node test**, such as A or node() in the examples above

Examples

Given a sample XML document

```
<?xml version="1.0" encoding="utf-8"?>
<wikimedia>
  <projects>
    <project name="Wikipedia" launch="2001-01-05">
      <editions>
        <edition language="English">en.wikipedia.org</edition>
        <edition language="German">de.wikipedia.org</edition>
        <edition language="French">fr.wikipedia.org</edition>
        <edition language="Polish">pl.wikipedia.org</edition>
        <edition language="Spanish">es.wikipedia.org</edition>
      </editions>
    </project>
    <project name="Wiktionary" launch="2002-12-12">
      <editions>
        <edition language="English">en.wiktionary.org</edition>
        <edition language="French">fr.wiktionary.org</edition>
```

```
        <edition language="Vietnamese">vi.wiktionary.org</edition>
        <edition language="Turkish">tr.wiktionary.org</edition>
        <edition language="Spanish">es.wiktionary.org</edition>
      </editions>
    </project>
  </projects>
</wikimedia>
```

The XPath expression

```
/wikimedia/projects/project/@name
```

Selects name attributes for all projects, and

```
/wikimedia//editions
```

Selects all editions of all projects, and

```
/wikimedia/projects/project/editions/edition[@language="English"]/text()
```

Selects addresses of all English Wikimedia projects (text of all edition elements where language attribute is equal to *English*). And the following

```
/wikimedia/projects/project[@name="Wikipedia"]/editions/edition/text()
```

Selects addresses of all Wikipedias (text of all edition elements that exist under project element with a name attribute of *Wikipedia*)

Use in schema languages

XPath is increasingly used to express constraints in schema languages for XML.

- The (now ISO standard) schema language Schematron pioneered the approach.

•A streaming subset of XPath is used in W3C XML Schema 1.0 for expressing uniqueness and key constraints. In XSD 1.1, the use of XPath is extended to support conditional type assignment based on attribute values, and to allow arbitrary boolean assertions to be evaluated against the content of elements.

•XForms uses XPath to bind types to values.

•The approach has even found use in non-XML applications, such as the constraint language for Java called PMD: the Java is converted to a DOM-like parse tree, then XPaths rules are defined over the tree.