KANO STATE POLYTECHNIC

SCHOOL OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

LECTURE NOTE ON

SWD 322

DATABASE DESIGN II

COURSE LECTURER:

KHADIJA MAHMOUD SANI

DATA AND ITS DIFFERENT STRUCTURES

In Computer Science, a data structure a particular way of organising and storing data in a computer such that it can be accessed and modified efficiently.

Structured vs unstructured data

A look into structured and unstructured data, their key differences and which form best meets your business needs.

Not all data are created equal; some are structured, but most of them are unstructured. Structured and unstructured data are sourced, collected and scaled in different ways and each one resides in a different type of database.

In this article, we will take a deep dive into both types so that you can get the most out of your data.

What is structured data?

Structured data—typically categorized as quantitative data—is highly organized and easily decipherable by machine learning algorithms. Developed by IBM® in 1974, structured query language (SQL) is the programming language used to manage structured data. By using a relational (SQL) database, business users can quickly input, search and manipulate structured data.

Pros and cons of structured data

Examples of structured data include dates, names, addresses, credit card numbers, among others. Their benefits are tied to ease of use and access, while liabilities revolve around data inflexibility:

Pros

- **Easily used by machine learning (ML) algorithms:** The specific and organized architecture of structured data eases the manipulation and querying of ML data.
- **Easily used by business users:** Structured data do not require an in-depth understanding of different types of data and how they function. With a basic understanding of the topic relative to the data, users can easily access and interpret the data.
- **Accessible by more tools:** Since structured data predates unstructured data, there are more tools available for using and analyzing structured data.

Cons

- **Limited usage:** Data with a predefined structure can only be used for its intended purpose, which limits its flexibility and usability.
- **Limited storage options:** Structured data are usually stored in data storage systems with rigid schemas (for example, "data warehouses"). Therefore, changes in data requirements necessitate an update of all structured data, which leads to a massive expenditure of time and resources.

Structured data tools

- **OLAP:** Performs high-speed, multidimensional data analysis from unified, centralized data stores.
- **SQLite: (link resides outside ibm.com)** Implements a self-contained, serverless, zero-configuration, transactional relational database engine.
- **MySQL:** Embeds data into mass-deployed software, particularly mission-critical, heavy-load production system.
- **PostgreSQL:** Supports SQL and JSON querying as well as high-tier programming languages (C/C+, Java, Python, among others.).

Use cases for structured data

- **Customer relationship management (CRM):** CRM software runs structured data through analytical tools to create datasets that reveal customer behavior patterns and trends.
- **Online booking:** Hotel and ticket reservation data (for example, dates, prices, destinations, among others.) fits the "rows and columns" format indicative of the pre-defined data model.
- **Accounting:** Accounting firms or departments use structured data to process and record financial transactions.

What is unstructured data?

Unstructured data, typically categorized as qualitative data, cannot be processed and analyzed through conventional data tools and methods. Since unstructured data does not have a predefined data model, it is best managed in non-relational (NoSQL) databases. Another way to manage unstructured data is to use data lakes to preserve it in raw form.

The importance of unstructured data is rapidly increasing. Recent projections (link resides outside ibm.com) indicate that unstructured data is over 80% of all enterprise data, while 95% of businesses prioritize unstructured data management.

Pros and cons of unstructured data

Examples of unstructured data include text, mobile activity, social media posts, Internet of Things (IoT) sensor data, among others. Their benefits involve advantages in format, speed and storage, while liabilities revolve around expertise and available resources:

Pros

- **Native format:** Unstructured data, stored in its native format, remains undefined until needed. Its adaptability increases file formats in the database, which widens the data pool and enables data scientists to prepare and analyze only the data they need.
- **Fast accumulation rates:** Since there is no need to predefine the data, it can be collected quickly and easily.
- **Data lake storage:** Allows for massive storage and pay-as-you-use pricing, which cuts costs and eases scalability.

Cons

- **Requires expertise:** Due to its undefined or non-formatted nature, data science expertise is required to prepare and analyze unstructured data. This is beneficial to data analysts but alienates unspecialized business users who might not fully understand specialized data topics or how to utilize their data.
- **Specialized tools:** Specialized tools are required to manipulate unstructured data, which limits product choices for data managers.

Unstructured data tools

- **MongoDB:** Uses flexible documents to process data for cross-platform applications and services.
- **DynamoDB:** (link resides outside ibm.com) Delivers single-digit millisecond performance at any scale through built-in security, in-memory caching and backup and restore.
- **Hadoop:** Provides distributed processing of large data sets using simple programming models and no formatting requirements.
- **Azure:** Enables agile cloud computing for creating and managing apps through Microsoft's data centers.

Use cases for unstructured data

- **Data mining:** Enables businesses to use unstructured data to identify consumer behavior, product sentiment and purchasing patterns to better accommodate their customer base.
- **Predictive data analytics:** Alert businesses of important activity ahead of time so they can properly plan and accordingly adjust to significant market shifts.
- **Chatbots:** Perform text analysis to route customer questions to the appropriate answer sources.

What are the key differences between structured and unstructured data?

While structured (quantitative) data gives a “birds-eye view” of customers, unstructured (qualitative) data provides a deeper understanding of customer behavior and intent. Let’s explore some of the key areas of difference and their implications:

- **Sources:** Structured data is sourced from GPS sensors, online forms, network logs, web server logs, OLTP systems, among others; whereas unstructured data sources include email messages, word-processing documents, PDF files, and others.
- **Forms:** Structured data consists of numbers and values, whereas unstructured data consists of sensors, text files, audio and video files, among others.
- **Models:** Structured data has a predefined data model and is formatted to a set data structure before being placed in data storage (for example, schema-on-write), whereas unstructured data is stored in its native format and not processed until it is used (for example, schema-on-read).
- **Storage:** Structured data is stored in tabular formats (for example, excel sheets or SQL databases) that require less storage space. It can be stored in data warehouses, which makes it highly scalable. Unstructured data, on the other hand, is stored as media files or NoSQL databases, which require more space. It can be stored in data lakes, which makes it difficult to scale.
- **Uses:** Structured data is used in machine learning (ML) and drives its algorithms, whereas unstructured data is used in natural language processing (NLP) and text mining.

What is semi-structured data?

Semi-structured data (for example, JSON, CSV, XML) is the “bridge” between structured and unstructured data. It does not have a predefined data model and is more complex than structured data, yet easier to store than unstructured data.

Semi-structured data uses “metadata” (for example, tags and semantic markers) to identify specific data characteristics and scale data into records and preset fields. Metadata ultimately enables semi-structured data to be better cataloged, searched and analyzed than unstructured data.

- **Example of metadata usage:** An online article displays a headline, a snippet, a featured image, image alt-text, slug, among others, which helps differentiate one piece of web content from similar pieces.
- **Example of semi-structured data vs. structured data:** A tab-delimited file containing customer data versus a database containing CRM tables.
- **Example of semi-structured data vs. unstructured data:** A tab-delimited file versus a list of comments from a customer’s Instagram.

DATA INTEGRITY

Data integrity refers to the accuracy, consistency, and completeness of data throughout its lifecycle. It's a critically important aspect of systems which process or store data because it protects against data loss and data leaks. Maintaining the integrity of your data over time and across formats is a continual process involving various processes, rules, and standards.

Why is Data Integrity Important?

Your organization is most likely flooded by large and complex datasets from many sources, both historical data and real-time streaming data. And you want to confidently make data-driven decisions that improve your business performance.

Incorrect or incomplete data can lead to bad decisions which can cost you significant time, effort, and expense. Plus, the loss of sensitive data–especially if it ends up in the hands of criminals–can mean enduring and wide-ranging negative impacts.

BENEFITS OF DATA INTEGRITY

- Supporting accurate data insights and decisions
- Protecting customers' and other data subjects' information such as personally identifiable information (PII), financial records and usage data
- Helping ensure regulatory compliance such as General Data Protection Regulation (GDPR)
- Ensuring quality in the product and/or service
- Ensuring safety and privacy of customers
- Increasing confidence of consumers to use online digital applications and tools
- Increasing the likelihood and speed of data recoverability in the event of a breach or unplanned downtime
- Protecting against unauthorized access and data modification
- Achieving and maintaining compliance more effectively

TYPES OF DATA INTEGRITY

To ensure integrity in both hierarchical and relational databases, there are the two primary types–physical integrity and logical integrity. Within logical integrity, there are four sub-categories: domain, entity, referential, and user-defined integrity. All are collections of rules and procedures which application programmers, system programmers, data processing managers, and internal auditors use to ensure accurate data.

PHYSICAL INTEGRITY refers to the rules and procedures which ensure the accuracy of data as it is stored and retrieved. Threats to physical integrity include external factors such as power outages, natural disasters and hackers and internal

factors such as storage erosion, human error or design flaws. Typically, the affected dataset is unusable.

LOGICAL INTEGRITY seeks to ensure that the data accurately makes sense in a specific context (whether it's "logical"). Logical integrity also has the challenge of human errors and design flaws. Thankfully, a dataset can be overwritten with new data and reused if it has a logical error. There are four topics of logical integrity as follows:

1. **Domain integrity** refers to the range of values such as integer, text, or date which are acceptable to be stored in a particular column in a database. This set of values (the "domain") has constraints which limit the format, amount, and types of data entered. All entries must be available in the domain of the data type. As shown in the example below, the entry for the number of Jean's orders is not an integer so it is out of domain. This would cause the database management system to produce an error.

Customer ID	Customer name	Age	Orders
44922945	Oliver Twist	34	21
30091920	James Gatz	42	9
75568215	Jean Finch	18	w#2@jk_1

Value is "out of domain" because it is not an integer.

2. **Entity integrity** uses primary keys to uniquely identify records saved in a table in a relational database. This prevents them from being duplicated. It also means they can't be NULL because then you couldn't uniquely identify the row if the other fields in the rows are the same. For example, you might have two customers with the same name and age, but without the unique identifier of the customer ID primary key, you could have errors or confusion when pulling the data.

Primary Key	Customer ID	Customer name	Age
	44922945	Oliver Twist	34
	30091920	James Gatz	42
Value cannot be NULL		James Gatz	42

3. **Referential integrity** refers to the collection of rules and procedures used to maintain data consistency between two tables. These rules are embedded into the database structure regarding how foreign keys can be used to ensure the data entry is accurate, there is no duplicate data, and, as in the example below, data which doesn't apply is not entered. You can see below how referential integrity is maintained by not allowing an order ID which does not exist in the order table.

First Table (Customers)

Customer ID	Customer name	Age	Order ID
44922945	Oliver Twist	34	498721009-87
30091920	James Gatz	42	448902161-53
75568215	Jean Finch	18	324163384-92

This value is not permitted because this value is not defined as a primary key in the Order ID table.

Second Table (Orders)

Order ID	Product ID	Order date
498721009-87	KF-62	03162022
448902161-53	KF-65	04112022

4. **User-defined integrity** acts as a way to catch errors which domain, referential and entity integrity do not. Here, you define your own specific business rules and constraints which trigger automatically when predefined events occur. For instance, you could define the constraint that customers must reside in a certain country to be entered into the database. Or, as in the example below, you might require that customers provide both first and last names.

Customer ID	Customer name	Age	Order ID
44922945	Oliver Twist	34	498721009-87
30091920	James Gatz	42	448902161-53
75568215	Jean	18	324163384-92

Value does not include a last name.

How to Ensure Data Integrity

There are a wide variety of threats to data integrity. And while most people imagine malicious hackers as the main threat, the majority of root causes are internal and unintentional, such as errors in data collection, inconsistencies across formats, and human error. You should build a culture of data integrity by:

- Educating business leaders on the risks
- Establishing a robust data governance framework
- Investing in the right tools and expertise.

ACID (atomicity, consistency, isolation, and durability)

What is ACID (atomicity, consistency, isolation, and durability)?

In transaction processing, ACID (atomicity, consistency, isolation, and durability) is an acronym used to refer to the four essential properties a transaction should possess to ensure the integrity and reliability of the data involved in the transaction. The acronym is commonly associated with relational database management systems (RDBMSs) such as MySQL or SQL Server, although it can apply to any system or application that processes transactions.

In computing, a transaction is a set of related tasks treated as a single action. Together, the tasks form a logical unit of work. For example, a bank customer might use a mobile app to transfer $200 from a savings account to a checking account. The customer logs into the app, enters the necessary information and clicks a button, which launches a "transfer" transaction.

The transaction consists of multiple steps, all invisible to the user. The steps might include verifying availability of funds in the savings account, deducting $200 from that account, adding $200 to the checking account, and verifying the final balance in each account to ensure the funds were transferred properly. All these tasks together represent the entire transaction; either they all succeed or none succeed.

If all the steps ran successfully, the transaction is considered complete and is committed to the managing system. The customer is notified that the funds were transferred from one account to the other. If any steps had failed, all completed steps would have been rolled back, and the data returned to its original state. In this case, the user would have received an error message.

Applying the ACID properties determines whether a transaction should be committed or rolled back, given the success or failure of each step in the transaction. If a transaction does not adhere to these properties, the integrity and reliability of the data can be called into question. For example, if the transfer transaction had failed

after deducting the $200 from the savings account but before adding the funds to the checking account, the customer's accounts would be short $200.

What are the ACID properties?

When data integrity and reliability are top considerations in a transaction processing system, the system will typically apply four properties to those transactions for ACID-compliance:

- **Atomicity.** A transaction is treated as a single atomic unit. All steps that make up the transaction must succeed or the entire transaction rolls back. If they all succeed, the changes made by the transaction are permanently committed to the managing system. Consider the transfer transaction example. For the transaction to be committed to the database, the $200 must be successfully deducted from the savings account and added to the checking account, and the funds in both accounts must be verified to ensure their accuracy. If any of these tasks fail, all changes roll back and none are committed.
- **Consistency.** A transaction must preserve the consistency of the underlying data. The transaction should make no changes that violate the rules or constraints placed on the data. For instance, a database that supports banking transactions might include a rule stating that a customer's account balance can never be a negative number. If a transaction attempts to withdraw more money from an account than what is available, the transaction will fail, and any changes made to the data will roll back.
- **Isolation.** A transaction is isolated from all other transactions. Transactions can run concurrently only if they don't interfere with each other. Returning to the transfer transaction example, if another transaction were to attempt to withdraw funds from the same savings account, isolation would prevent the second transaction from firing. Without isolation, it might be possible for the second transaction to withdraw more funds than are available in the account after the first transaction was completed.
- **Durability.** A transaction that is committed is guaranteed to remain committed -- that is, all changes are made permanent and will not be lost if an event such as a power failure should occur. This typically means persisting the changes to nonvolatile storage. If durability were not guaranteed, it would be possible for some or all changes to be lost, affecting the data's reliability.

Systems that support transaction processing employ various methods to ensure that each transaction is ACID-compliant. For example, SQL Server includes integrity constraints, such as primary key, foreign key, unique and check, to achieve data consistency. The system also supports isolation levels, such as read committed, read uncommitted and snapshot, to provide the necessary isolation for each transaction. These features are built into the database system so they can be incorporated easily into an application's backend database, although they need to be carefully implemented to ensure maximum ACID compliance.

Achieving ACID compliance for distributed transactions is more complicated than those based on a single location.

DATABASE INDEX

Indexes are data structures that can increase a database's efficiency in accessing tables. Indexes may not necessarily be required; the database can function properly without them, but query response time can be slower.

Every index is associated with a table and has a key, which is formed by one or more table columns. When a query needs to access a table that has an index, the database can decide to use the index to retrieve records faster.

Question: What is a SQL Index?

A SQL index is a database structure that speeds up data retrieval operations by providing quick access to table rows. It works like a book's index, allowing the database to find data without scanning the entire table.

How Does an Index Work?

If you've ever used a book index, you'll understand immediately how a database index works. In a book index, we first search the index for the topic we want. We get the page number where that topic is found in the book, and we go to this page. It's very simple.

Index

When the database searches in an index, the first step is to find the key value in the index. Every key value is stored with a pointer to the record in the table associated with this key value. Then, when the key value is found, the database follows that pointer and directly reads the record(s) from the table.

Suppose we table with 40 million records and a schema like the following:

Ssn	first_name	last_name	zip_code
543678282	John	Smith	ZZ123459
512345656	Igor	Chernosky	ZZ123459
432583434	Mary	Connery	AF432112
474812222	Sue	Leclerc	AW543112

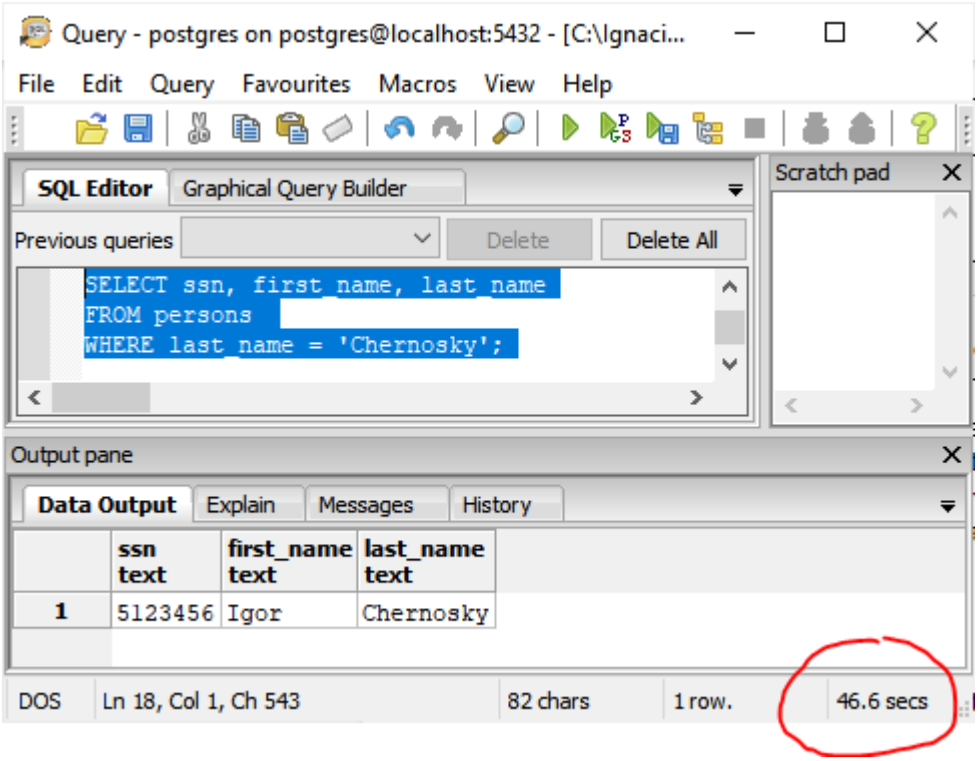
We run this query:

```
SELECT ssn

FROM persons

WHERE last_name = 'Chernosky'
```

To get the results, the database must do a full sequential scan of the table **persons**, which means that the database must read every record on the table and verify if the WHERE condition last_name = 'Chernosky' for this record is true or false. As this table has 40 million rows, the database is carrying out more than a few operations. On my local machine, this query takes 46 seconds!



How to Add an Index to a Table

At this point, we have a really slow query – it takes 46 seconds to return just one record. We can try to add an index to the table **persons** using the column last_name as the index key. The idea is to improve query performance by avoiding a full sequential scan of the entire table.

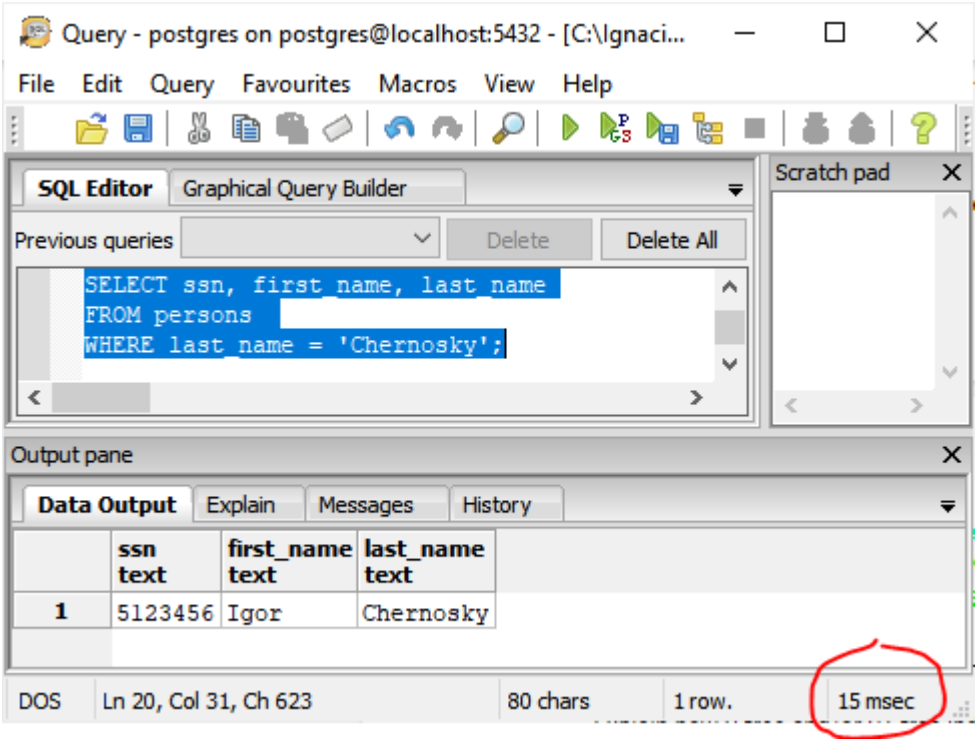
Here's how to create such an index:

```
CREATE INDEX ix1_test ON persons ( last_name )
```

The syntax for index creation is simple. We need a CREATE INDEX statement specifying:

- The index name (here, ix1_test).
- The associated table (**persons**).
- The column to use as the index key (last_name).

After creating the index and executing the previous query a second time, the database will do an index search for the last_name = 'Chernosky' condition instead of a full sequential scan. Let's see how much time it takes.



With the new index, the same query takes 15 milliseconds – 30,000 times faster! The reason behind this huge performance increase is using the index to look for last_name = 'Chernosky' instead of doing a full sequential scan. Remember, an index's process is similar to the process of using a book index. Imagine how much time you'd need to find a specific topic if you decided to search for it by reading the entire book, page by page!