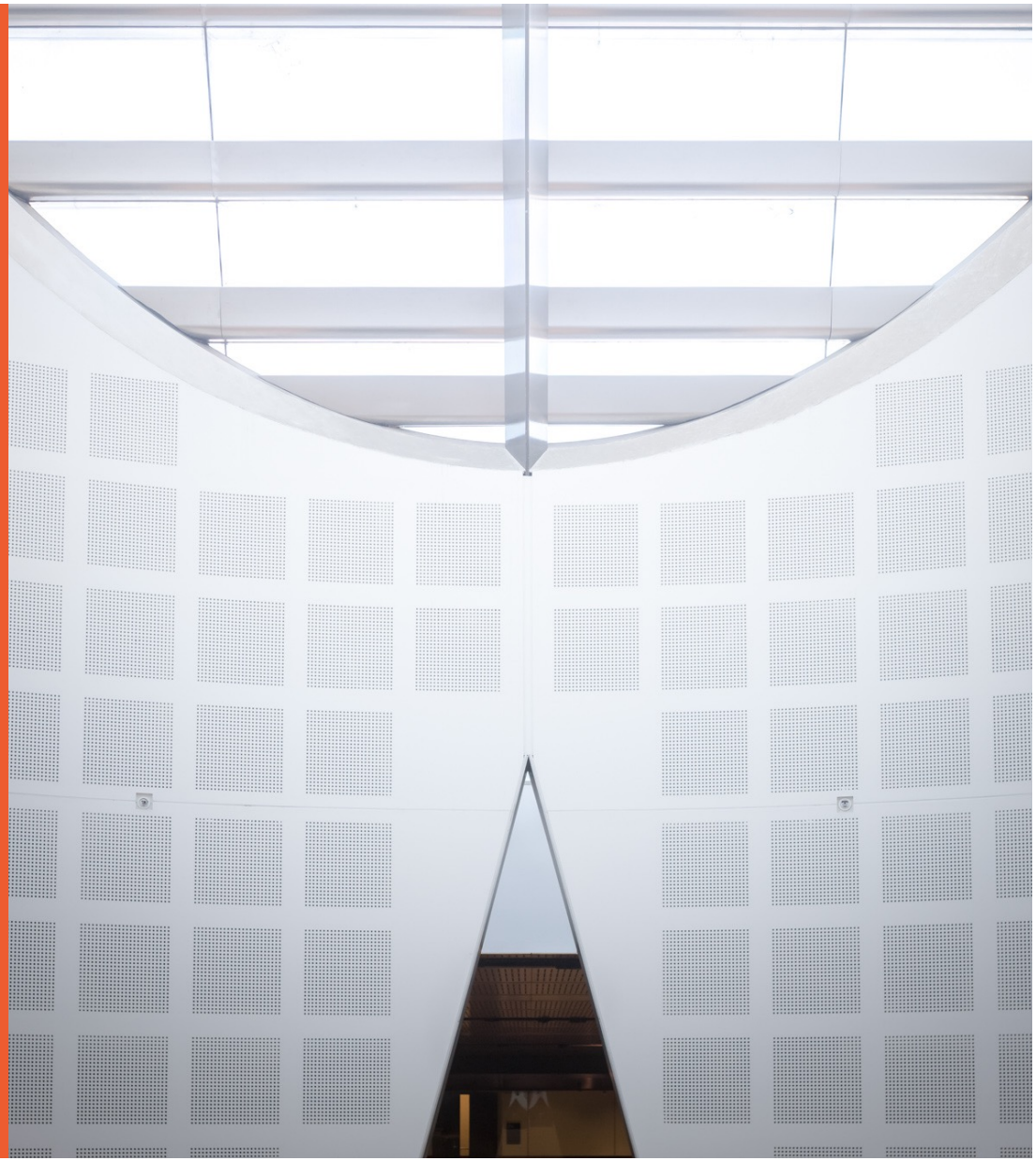
COMP5339: Data Engineering

Week 3: Databases, Data Warehouses and Data Lakes

Presented by
Uwe Roehm
School of Computer Science

THE UNIVERSITY OF SYDNEY



Data Acquisition – Where does data come from?

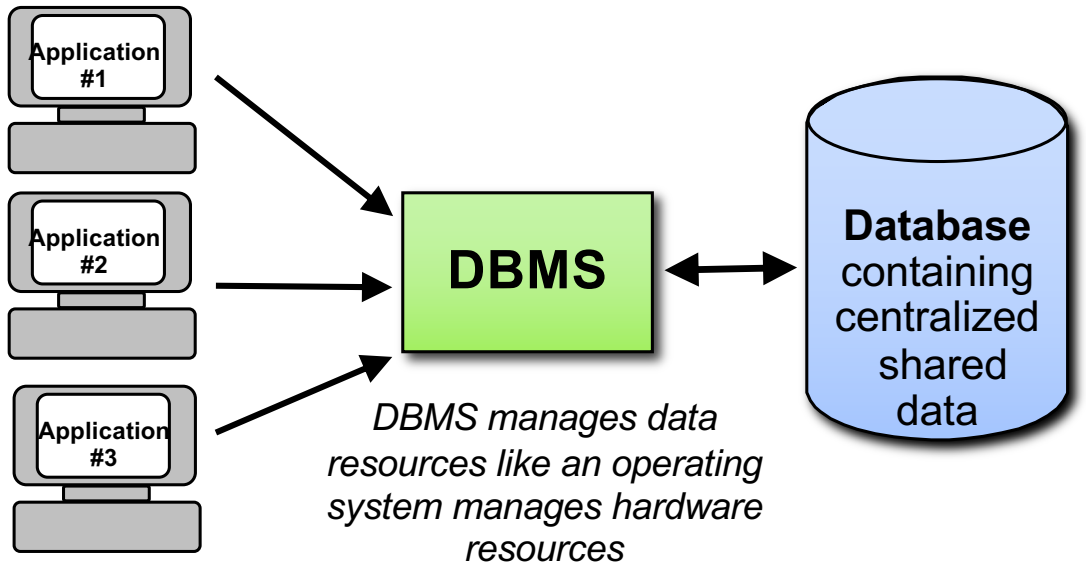
- File Access
  - Existing files / datasets of an enterprise;
    or download from an online web/data server (e.g. data.gov.au)
  - Typical exchange formats: CSV, Excel, sometimes XML or JSON
  - Also unstructured files such as text or images
- Database Access / Application Databases
- Programmatically / APIs
  - E.g. scraping the web (HTML)
  - or using APIs of Web Services or IoT devices (XML/JSON)
- Change Data Capture (CDC) / Logs
- Messages and streams, Publish/Subscribe

The Database Approach

The Database Approach

DBMS is the interface

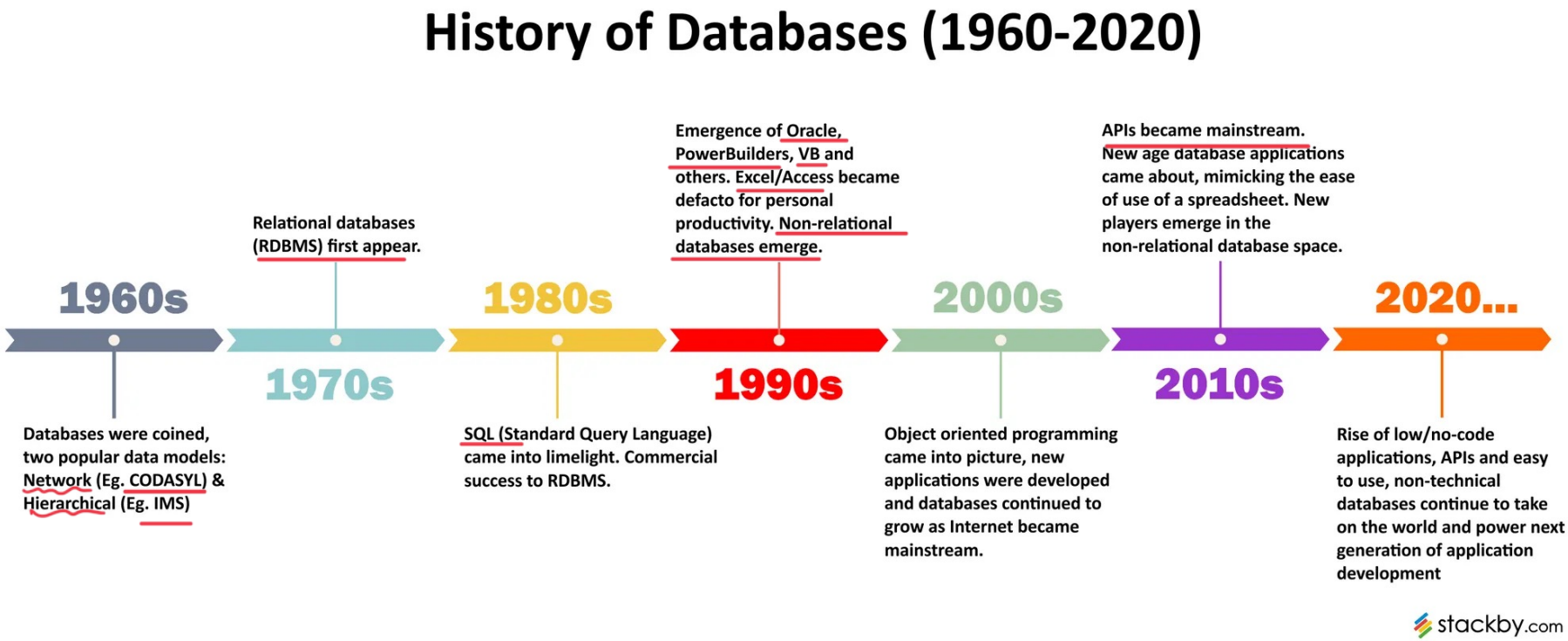
- Central repository of shared data
- Data is managed by software: Database Management System (DBMS)
- Data is accessed by applications or users, always through the DBMS



Advantages of Database-approach to Data Engineering

- Data is managed, so **quality can be enforced by the DBMS**
  - Improved Data Sharing
    - Different users get different views of the data
    - Efficient concurrent access
  - Enforcement of Standards
    - All data access is done in the same way
  - Integrity constraints, data validation rules
  - Better Data Accessibility/ Responsiveness
    - Use of standard query language (SQL)
  - Security, Backup/Recovery, Concurrency
    - Disaster recovery is easier

History of Databases



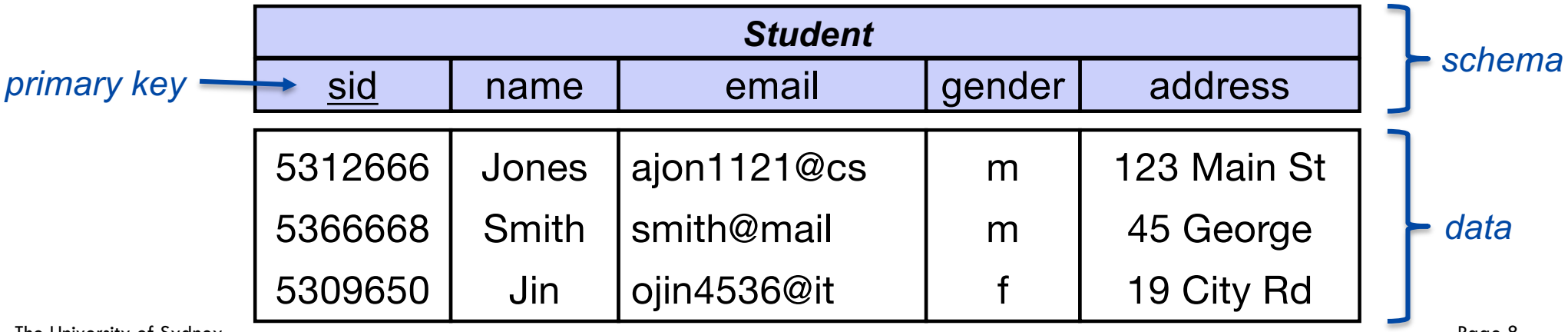
Relational Databases

- Relational data model is the most widely used model today
  - Main concept: **relation**, basically a table with rows and columns
  - Every relation has a **schema**, which describes the columns, or fields

- This sounds like a spreadsheet, but as we will see, it has some differences

Relational Database Systems

- store data in **tables** as **rows** with multiple **attributes**
- rows *of the same format* form a 'table' (**relation: a set of tuples**)
  - Every relation has a **schema**, which describes the columns, or fields, and their types
- A relational database is a collection of such tables
  (which typically are related to each other by **key** attributes)
- Example:



Primary Key

- A primary key is a unique attribute (or combination of attributes) which the database uses to identify a row in a table.
- It is a unique (typically auto-incrementing) ID which is NEVER NULL
  - NULL has the special meaning in databases of "unknown" or "not given"

Author		
authorId	**given_name**	**family_name**
1	Charles	Dickens
2	Virginia	Woolf

Foreign Key

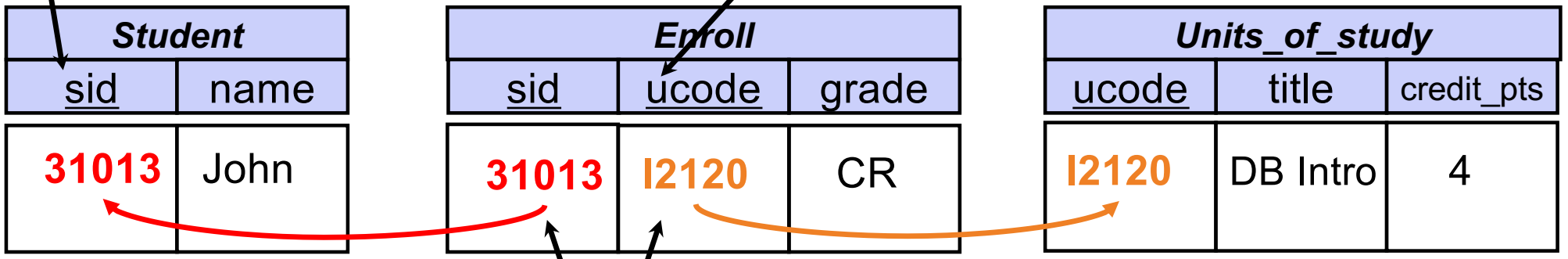
- When we need to refer to a record in a separate table we reference its ID as a foreign key.
- A **foreign key** is defined in a second table, but it refers to the primary key or a unique key in the first table.

Books		
bookID	**title**	**authorID**
1001	Orlando	2
1002	David Copperfield	1

Example: Relational Keys

Primary key identifies each tuple of a relation.

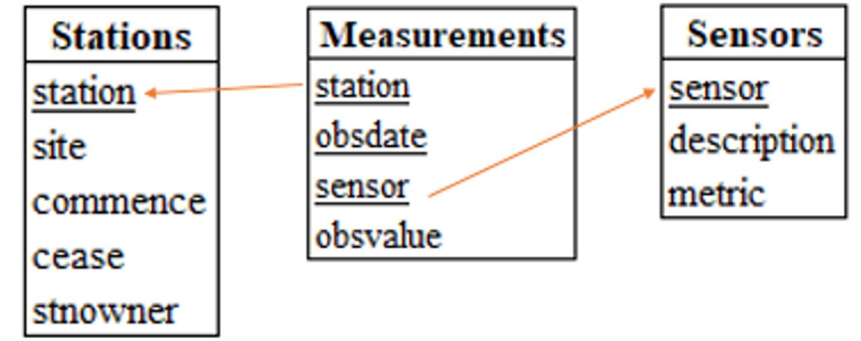
Composite Primary Key consisting of more than one attribute.



Foreign key is a (set of) attribute(s) in one relation that `refers' to a tuple in another relation (like a `logical pointer').

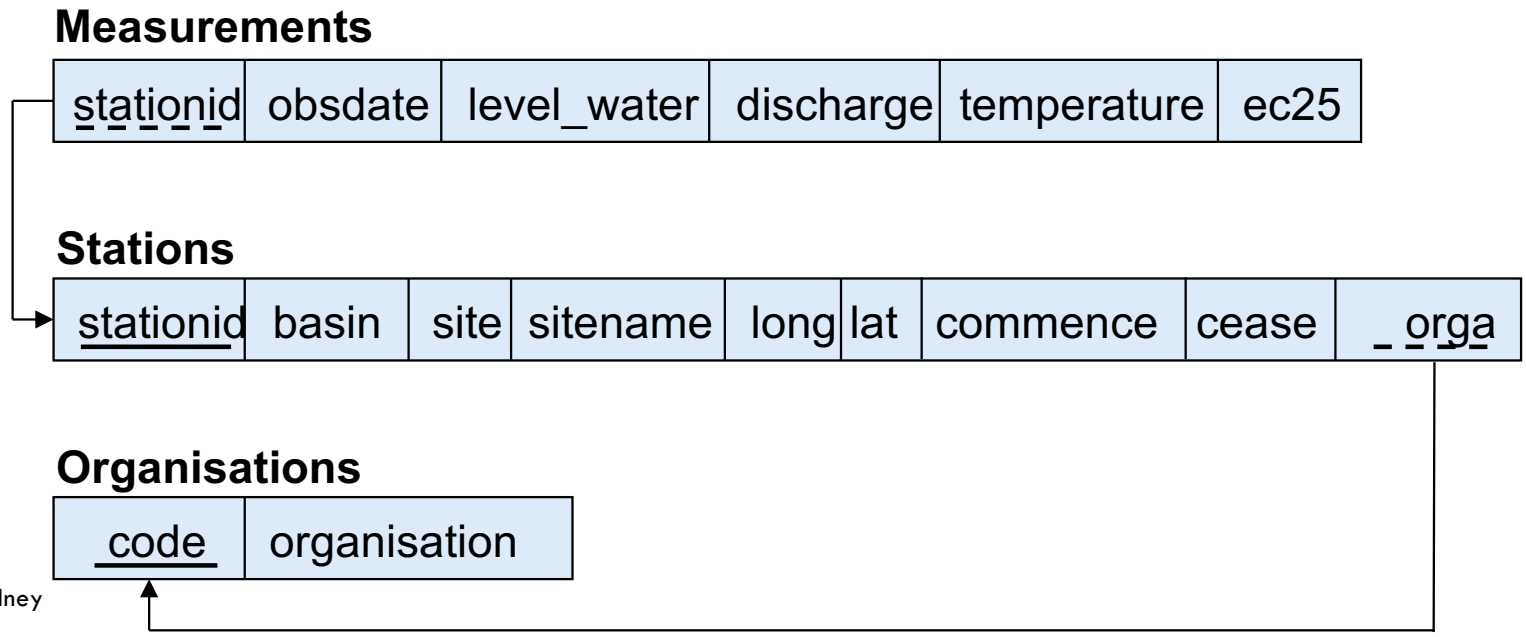
Schema Diagrams

- A *normalised* relational database tries to avoid redundancies
  - Every fact is ideally stored only once
  - That's some difference to spreadsheet where lots of data gets repeated and then tends to become inconsistent
- Different graphical notations used to visualise a schema
  - Also: Entity-Relationship-Diagrams



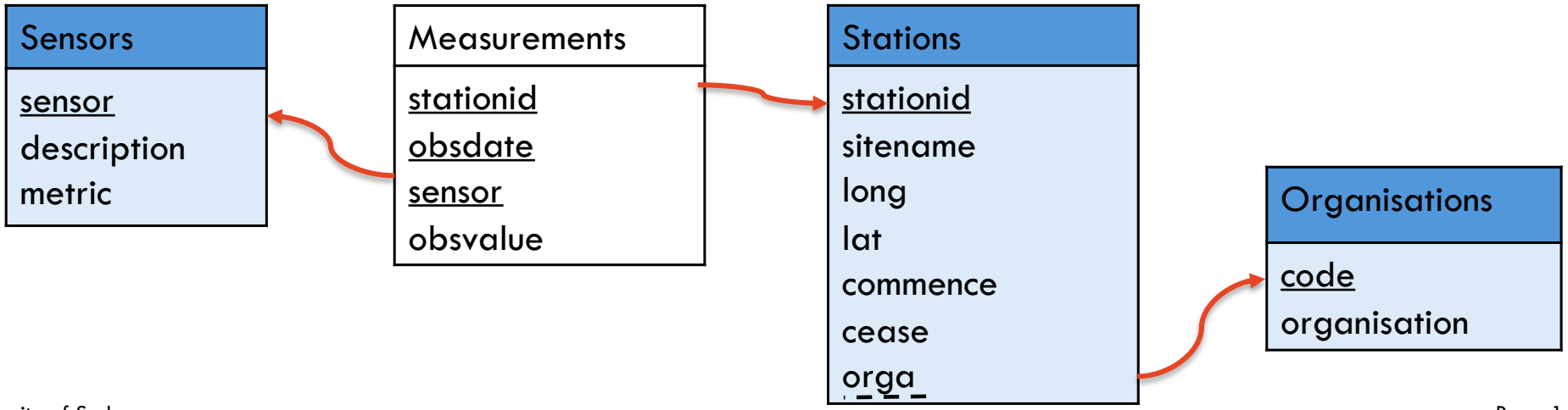
WaterInfo Example as a Relational Database – Option 1

- Design Option 1: Straight-forward 1:1 mapping
  - Because a spreadsheet is in principle a table, we can always map it 1:1
  - Some problems though: e.g. the Station <-> Basin+Site mapping
  - A lot of NULL values for any missing measurement or if no sensor



Modeling our Water Data Set – Option 2

- Design Option 2: Normalised Relational Schema
  - measurements are **facts**, other data describes the **dimensions**
  - measurement entries get 'folded' into separate rows of the fact table
  - allows us to avoid NULLs as much as possible, but hard to read

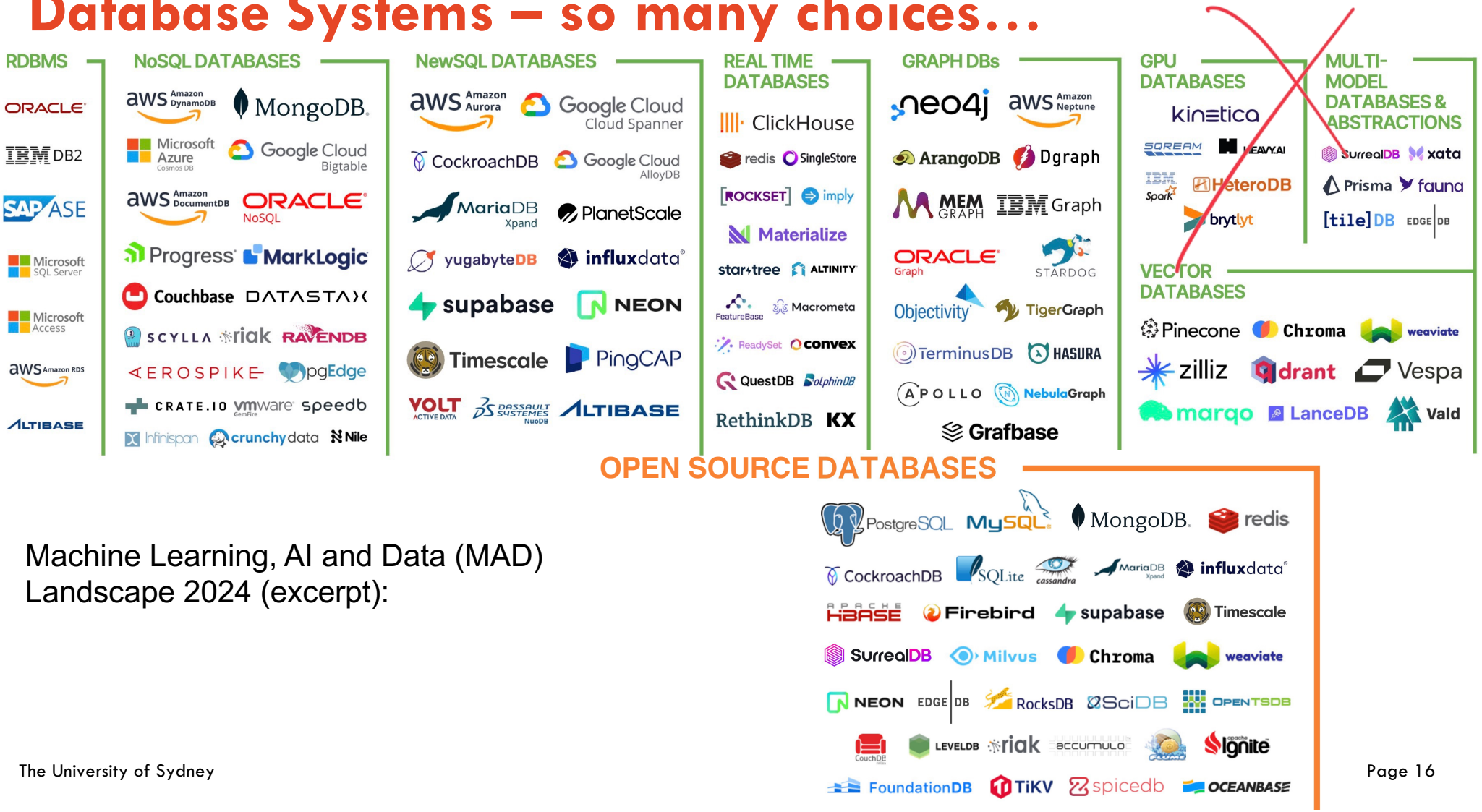


Various Database Systems

- Database Management System (DBMS)

- Relational Databases vs. specialized / NoSQL databases
- Analytical vs. operational databases
- Commercial vs. open-source
- Traditional disk-based vs. main-memory vs. cloud-based

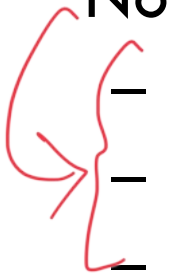
Database Systems – so many choices…



Machine Learning, AI and Data (MAD) Landscape 2024 (excerpt):

Database Systems – The Main Players

- Commercial DBMS
  - IBM DB2
  - Oracle
  - Microsoft SQL Server
  - SAP ASE
- Analytical DBMS
  - Snowflake
  - Teradata
  - Vertica
- OpenSource DBMS
  - MySQL
  - PostgreSQL
  - SQLite
- NoSQL Systems
  - MongoDB
  - Couchbase
  - Redis



See also "Database of Databases" at https://dbdb.io

Specialised DBMS Examples

- NoSQL ‘Document’ Databases for semi-structured data (JSON)
  - MongoDB, Couchbase
- Graph Databases
  - neo4J
- Time Series Databases
  - InfluxData, TimescaleDB
- Visualisation + in-memory Databases
  - tableau

Work with DBMS

- Many options for data management systems
  - Choice depends on existing infrastructure, capabilities/special needs, budget, taste
- Choice: build solutions that are DBMS-agnostic or that are tightly integrated?
- Pushing compute into the DBMS layer is possible and can help to scale – but can mean reduced choices and potential 'lock-in' to one technology

Databases as a Data Source

Data Gathering

- Databases are also a common source for data analysis
- Two approaches:
  - push queries into DBMS, let it evaluate there, and just retrieve query result
  - extract large chunks or even all data, and then analyse outside DBMS

- In the following, we will use PostgreSQL as an example platform; principles apply to any database system though.

Example: Accessing Databases from Python

- First, we need to import the database driver, e.g. psycopg2, then **connect** to database (here Postgresql)
  - For this, you also need some login credentials for your database
- As this is all quite system specific, best factored into a separate connect() function

```python
from sqlalchemy import create_engine, text, inspect
import psycopg2
import json
import pandas as pd

credentials = "Credentials.json"

def pgconnect():
    with open(credentials) as f:
        db_conn_dict = json.load(f)
        host        = db_conn_dict['host']
        db_user     = db_conn_dict['user']
        db_pw       = db_conn_dict['password']
        default_db  = db_conn_dict['user']
        try:
            engine = create_engine(f'postgresql+psycopg2://{db_user}:{db_pw}@{host}/{default_db}', echo=False)
            conn = engine.connect()
            print('Connected successfully.')
        except Exception as e:
            print('Unable to connect to the database.')
            print(e)
            engine, conn = None, None
        return engine, conn
```

Querying PostgreSQL from Python

- How to execute an SQL statement on a given connection 'conn'
  - Pandas supports loading a Dataframe from an RDBMS using **read_sql()**
  - This utility function, query(), helps to distinguish between executing commands and queries with results that are returned as DataFrame

```
def query(conn, sqlcmd, args=None, df=True):
    result = pd.DataFrame() if df else None
    try:
        if df:
            result = pd.read_sql(sqlcmd, conn, params=args)
        else:
            result = conn.execute(sqlcmd, args).fetchall()
            result = result[0] if len(result) == 1 else result
    except Exception as e:
        print("Error encountered: ", e, sep='\n')
    return result
```

Querying PostgreSQL from Python (cont'd)

- Example: Retrieving some data from the database

```
# connect to your database
engine, conn = pgconnect()

# prepare SQL statement
query_stmt = "SELECT * FROM Sensor"

# execute query and show first few rows of result
query_result = query(conn, query_stmt)
print(query_result.head())

# prepare another SQL statement including a placeholder
query_stmt2 = "SELECT COUNT(*) FROM Measurement WHERE station=%(station)s"

#execute query and print result
param = {'station':'409204C'}
query_result = query(conn, query_stmt2, param)
print(query_result)

#cleanup
conn.close()
```

Example *range query*: query all rows of a table

parameter binding

Example *point query*: query a specific row

Data Visualisation from SQL in Python

```
%matplotlib inline
import matplotlib.pyplot as plt

# connect to your database
engine, conn = pgconnect()

# prepare (multiline) SQL statement:
# 'How many sensor measurements did each site report today?'
query_stmt = '''SELECT sitename, COUNT(*) AS sensor_readings
                FROM Measurement JOIN Station USING (station)
                WHERE date = %(obsdate)s
                GROUP BY sitename'''

# execute query
param = {'obsdate':'2004-12-31'}
query_result = query(conn, query_stmt, param)
print(query_result)

# visualise result
plt.bar(query_result['sitename'], query_result['sensor_readings'])
plt.title('Sensor Readings per Station')
plt.xlabel('Measurement Station')
plt.ylabel('Number of Sensor Readings')
plt.grid()

# cleanup
conn.close()
```

Let the database do all group-by query work

Visualise result as bar chart

Bulk-Load Data from Storage

- In previous slides, we checked how we can selectively load data from a database using SQL queries
  - Advantage: pushes work into DBMS, let it help us e.g. with join processing or using indexes
- Alternatively: Load table in bulk from DBMS, then process in Python
  - Example: df = **pd.read_sql_table**('measurements', conn)
  - Advantage: Easy
  - Disadvantage: DBMS cannot help optimize; needs to fit into main memory
- The more advanced the DBMS, the more functionality you can delegate to it
  - Can be a massive scalability advantage; as long as with SQL, is also system-agnostic
  - Advanced: User-defined functions / stored procedures, but then tends to lock-in to platform

Databases as Sink

Data Storage

- So far we looked at the DBMS as the source for data
- But databases are also a common sink for storing data
- Main approaches:
  - External GUI or command line tools
  - Programmatically using INSERT statements
  - Semi-automated using a mapper framework, such as Pandas' to_sql()

Approach 1: External GUI / Command Line Tools

- For example, Postgresql 's psql provides a command to load data directly from a CSV file into a database table
  - \COPY tablename FROM filename CSV [HEADER] [NULL '...']
  - Many further options
  - Try \help COPY
- Similar: SQL browsers like **pgadmin** often have an 'Import CSV…" function
- Pros:
  - Relatively fast and straight-forward
  - No programming needed
- Cons:
  - Only 1:1 mapping of CSV to tables; no data cleaning or transformation
  - Stops at the first error…

Approach 2: Programmatically using SQL

- We also can load data with a sequence of SQL statements that are generated by a script or code
  - Either generated by a Unix script and then executed as a SQL file:

```
awk 'BEGIN {FS=","}
    {print "INSERT INTO Stations VALUES ("$1 substr($2,1,3)
","$3","$4","$5","$6","$7","$8");"}'
    Stations_raw.csv > Stations.sql
```

  - or programmatically in eg. Python:

```
insert_stmt = "INSERT INTO Measurements VALUES (%(station)s, CURRENT_DATE,
%(sensor)d, %(value)d)"
for row in data:
    params = {'station': row['stationid'], 'sensor': row['sensor'], 'value':
row['obsvalue'] }
    success = query(conn, insert_stmt, param, False)
```

Approach 2: Programmatically using SQL (cont'd)

- We also can load data with a sequence of SQL statements that are generated by a script or code

- Pros
  - Can be optimized for your specific data and use case
  - Full flexibility regarding data cleaning and data transformations
- Cons
  - Development time
  - Changing data can require schema (or transformation) changes, which need to be done manually

Approach 3: Pandas Loading Code

- Built-in functionality in Pandas: to_sql()
  - Obtain data as DataFrame df (eg. df = pd.read_csv(file))
  - Execute df.to_sql(table, conn) on a database connection conn
- This creates a new table based on the header row from the Dataframe

- Can also combine with creating your own table with own integrity constraints
  - Connect to the database
  - Define a string with the appropriate `CREATE TABLE` command
  - Use conn.execute(string) to do this in the dbms
  - Execute df.to_sql(table, conn, if_exists="append") on db connection

- Pros: flexibility; use pandas data cleaning and transformation tools
- Cons: needs to be hand-coded

Approach 3: Pandas Loading Code Example

- Example: Creating a table and loading some data

```
# 1st: login to database
engine, conn = pgconnect()

# if you want to reset the table
conn.execute(text("DROP TABLE IF EXISTS organisations CASCADE"))

# 2nd: ensure that the schema is in place
organisation_schema = """CREATE TABLE IF NOT EXISTS organisations (
                         code         CHAR(3) PRIMARY KEY,
                         organisation VARCHAR(80)
                     )"""
conn.execute(text(organisation_schema))

# 3rd: load data using pandas
data = pd.read_csv('Organisations.csv')
data.columns = data.columns.str.lower()
# print(data)
data.to_sql(name="organisations", con=conn, if_exists='append', index=False)
conn.commit()
```

utility function which we encapsulates db connection handling

Actual data storage

Schema Design

- When loading data with either external tools, or using automatic mappers such as Pandas' pd.to_sql(), a schema will be generated automatically based on the attributes in the CSV file or the Dataframe
  - Such schemas have no foreign key relationships, nor any integrity constraints.
  - They are also only as 'normalized' as the original CSV/Dataframe is…

- Or: design and create your own schema first into which you then load the data
  - SQL includes DDL (Data Definition Language) for this such as `CREATE TABLE`
  - More control and opportunity to choose data types and to add integrity constraints
  - But can make loading raw data a nightmare as data now has to pass constraints
  - Compromise: Normalised schema, code-based loading

Example: WaterInfo Schema DDL

```
CREATE SCHEMA WaterInfo;
CREATE TABLE Organisations (
    code         CHAR(3)  PRIMARY KEY,
    organisation VARCHAR(80)
);
CREATE TABLE Stations (
    stationid INT         PRIMARY KEY,
    sitename  VARCHAR(40) NOT NULL,
    commence  DATE,
    cease     DATE,
    orga      CHAR(3) REFERENCES Organisations(code)
);
CREATE TABLE Sensors (
    sensor      VARCHAR(6) PRIMARY KEY,
    description TEXT,
    metric      TEXT
);
```

```
CREATE TABLE Measurements (
    stationid INT    REFERENCES Stations(stationid),
    obsdate   DATE,
    sensor    VARCHAR(6) REFERENCES Sensors(sensor),
    obsvalue  NUMERIC,
    PRIMARY KEY (stationid, obsdate, sensor),

);
```

Non-relational Databases

- many data management tasks are naturally handled with tables, but..
- other possibilities:
  - key/value
  - time series
  - spatial
  - .. and others
- ingestion/loading solutions are similar, but with differences

OLAP

THE UNIVERSITY OF SYDNEY

Operational Systems vs Analytical Systems

- Operational systems consist of the backend services and data infrastructure where data is created, for example by serving external users.
  - An operational system typically looks up a small number of records by some key (this is called a **point query**).
  - Records are **inserted, updated, or deleted** based on the user's input.
  - This access pattern is known as online transaction processing (**OLTP**).

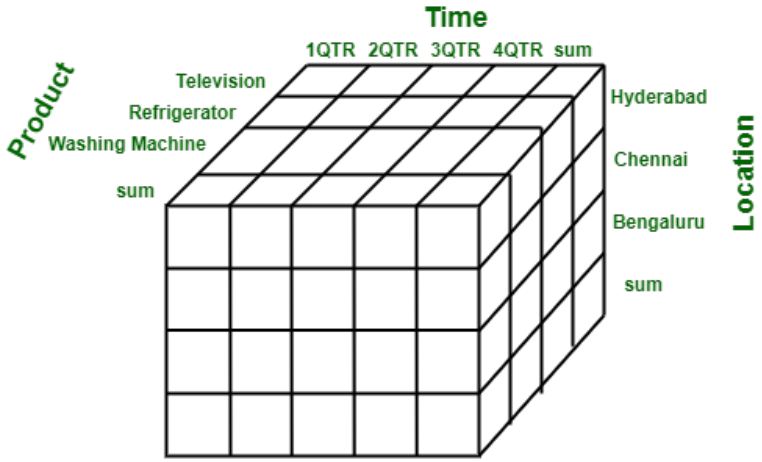
modify

- Analytical systems serve the needs of business analysts and data scientists.
  - They contain a **read-only** copy of the data from the operational systems.
  - An analytic query usually scans over a **huge number of records,** and calculates aggregate statistics (such as count, sum, or average) rather than returning the individual records to the user.
  - This access pattern is known as online analytical processing (**OLAP**).

not modify

OLAP

- OLAP enables analysts to extract and view data from different points of view.
  - E.g., what was the total revenue of each of our stores in January?
- Characteristics:
  - Interactive and "online" queries based on spreadsheet-style operations and "multidimensional" view of data (called **cube**).
  - Complex SQL queries
  - Trend analysis.
- Use Cases: Business reporting, data mining, financial reporting.



https://media.geeksforgeeks.org/wp-content/uploads/20210625184515/GFGMultidimensionaldataModel.png

OLAP In Practice

- Business analysts perform several basic analytical operations (e.g., roll up, drill down, slice, dice, pivot) with a multidimensional online analytical processing (MOLAP) cube.

- In **roll up**, the OLAP system summarises the data for specific attributes. It shows less-detailed data.
  - E.g., you might view product sales according to New York, California, London, and Tokyo. A roll-up operation would provide a view of the sales data based on countries, such as US, UK, and Japan.

- **Drill down** is the opposite of the roll-up operation. Business analysts move downward in the concept hierarchy and extract the required details.
  - E.g., they can move from "viewing sales data by years" to "visualising it by months."

year → month

OLAP In Practice (cont')

- Data engineers use the **slice** operation to select a single dimension from a cube, creating a sub-cube.
  - E.g., a MOLAP cube sorts data according to months, products and cities. By slicing the cube, data engineers can create a spreadsheet-like table consisting of products and cities for a specific month.

- Data engineers use the **dice** operation to create a smaller sub-cube from an OLAP cube. They determine the required dimensions and build a smaller cube from the original hypercube.

- The **pivot** operation involves rotating the OLAP cube along one of its dimensions to get a different perspective on the multidimensional data model.
  - E.g., switching rows and columns in a report from product-wise sales to region-wise sales.

OLTP and OLAP Example

OLTP

Start Transaction: Initiate a transaction to ensure atomicity.

Check Account Balance: Query the sender's account balance/Ensure sufficient funds are available.

Debit Amount: Subtract the transfer amount from account/ Update the sender's account balance.

Credit Amount: Add the transfer amount to the receiver's account/ Update the receiver's account balance.

Commit Transaction: If all operations are successful, commit the transaction to make changes permanent.

Rollback (If Error): If any operation fails, rollback the transaction to maintain data integrity.

End Transaction: Complete the transaction process.

OLAP

Extract Data: Select sales records from the data warehouse for the past year.

Transform Data: Group sales records by product category.

Aggregate Data: Calculate the total revenue for each product category by summing the sales amounts.

Sort Data: Sort the results by revenue in descending order for reporting.

Load Data: Load the aggregated data into a reporting table or dashboard for analysis.

OLTP vs OLAP Summary

OLTP (Online **Transaction** Processing):

- Manages real-time transaction data.
- Focuses on fast query processing and maintaining data integrity.
- Example: Bank transactions, order entry systems.

OLAP (Online **Analytical** Processing):

- Designed for complex queries and data analysis.
- Supports multidimensional data analysis.
- Example: Data warehousing, business reporting.

Data Warehouse

Data Warehouse

- At first, the same databases were used for both transaction processing and analytic queries.
  - SQL turned out to be quite flexible in this regard: it works well for both types of queries.

- In the late 1980s and early 1990s, there was a trend for companies to stop using their OLTP systems for analytics purposes, and to run the analytics on a separate database system instead.
  - This separate database was called a data warehouse.

Data Warehouse

- A large enterprise may have dozens, even hundreds, of online transaction processing systems
  - E.g., systems powering the customer-facing website, controlling point of sale (checkout), tracking inventory in warehouses, managing suppliers, administering employees, and etc.
  - Each of these systems is complex and needs a team of people to maintain it, so these systems end up operating mostly independently from each other.
- It is usually undesirable for business analysts and data scientists to directly query these OLTP systems, for several reasons:
  - The data of interest may be spread across multiple operational systems, making it difficult to combine those datasets in a single query (a problem known as **data silos**).
  - The kinds of schemas and data layouts that are good for OLTP are less well suited for analytics (e.g., star and snowflake schema).
  - Analytic queries can be quite expensive, and running them on an OLTP database would impact the performance for other users.

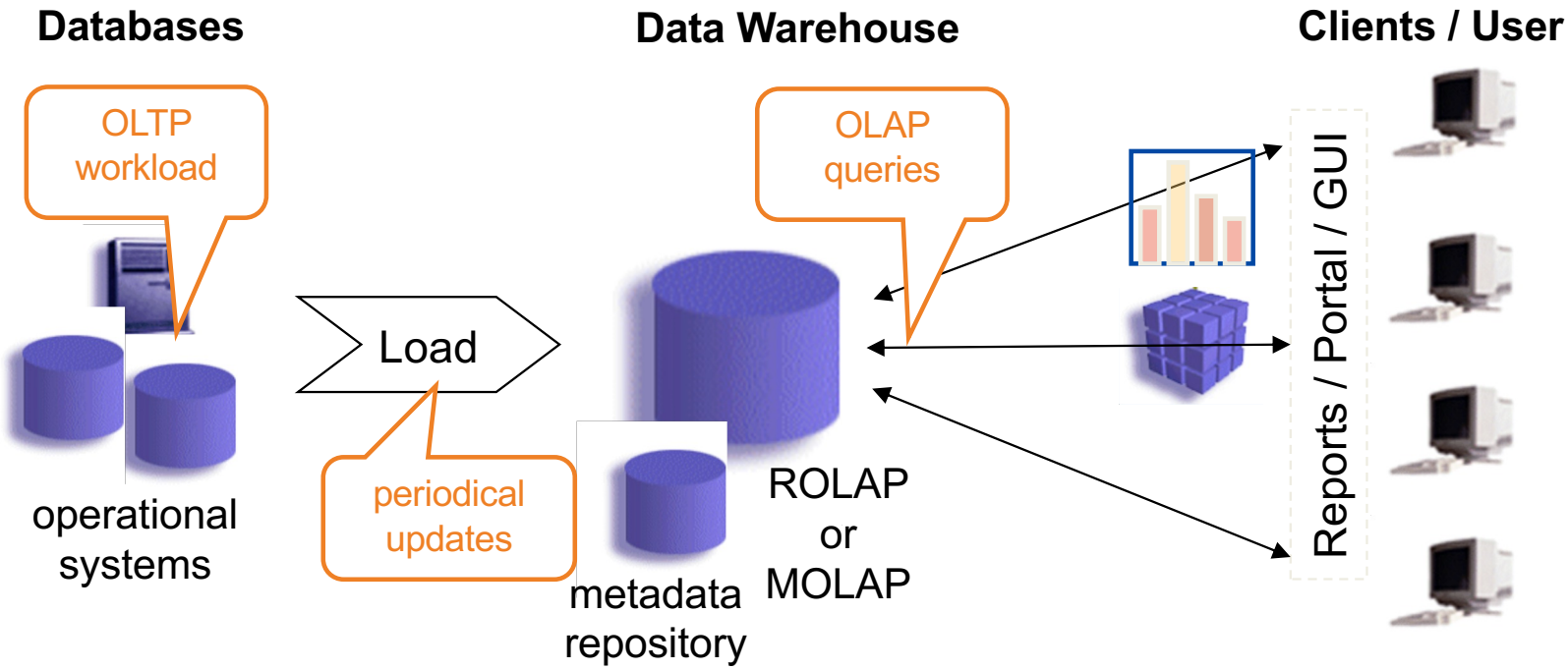
Data Warehouse

- A data warehouse is centralised repository optimised for analysis where data from different sources are consolidated in a single **structured** format.
  - It is a separate database that analysts can query to their hearts' content, without affecting OLTP operations.

- Characteristics:
  - Designed for complex queries and analysis
  - Data is cleaned, transformed
  - Typically uses a star or snowflake schema

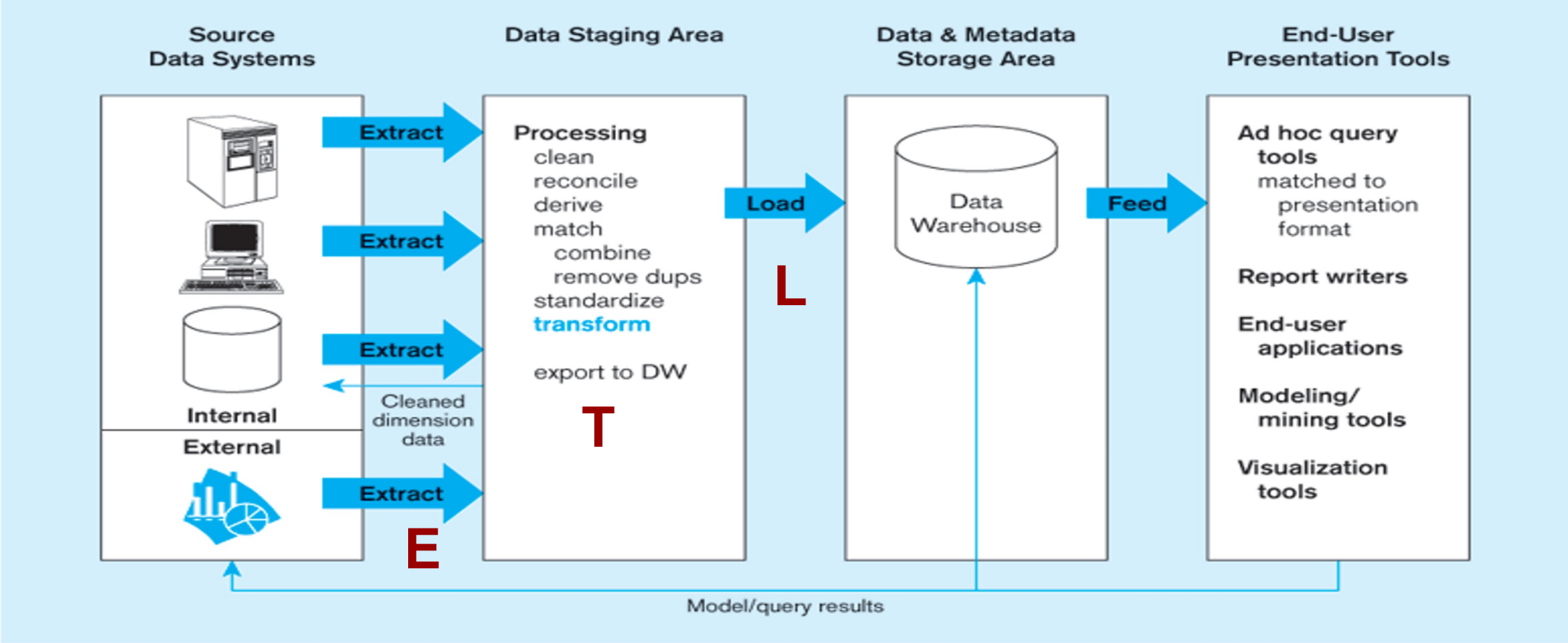
OLTP vs OLAP

The data warehouse contains a **read-only** copy of the data in all the various OLTP systems in the company.



Populating a Data Warehouse: ETL Process

– **ETL Process:** Capture/Extract - Data Cleansing - Transform - Load



Issues in Data Warehousing

- Semantic Integration: When getting data from multiple sources, must eliminate mismatches, e.g., different currencies, schemas.
  - E.g. *schema* used in different DBMSs for the same data might differ
    - Attribute names: SSN vs. Ssnum
    - Attribute domains: Integer vs. String
  - *semantics* might be different
    - Summarizing sales on a daily basis vs. summarizing sales on a monthly basis
- Heterogeneous Sources: Must access data from a variety of source formats and repositories.
  - Replication capabilities can be exploited here.
- Load, Refresh, Purge: Must load data, periodically refresh it, and purge too-old data.
  - E.g. Data Cleaning: Removing errors and inconsistencies in data
- Metadata Management: Must keep track of source, loading time, and other information for all data in the warehouse.

Database vs Data Warehouse

	Database (OLTP)	Data Warehouse (OLAP)
Main read pattern	Point queries (fetch individual records by key)	Aggregate over large number of records
Main write pattern	Create, update, and delete individual records	Bulk import (ETL) or event stream
Human user example	End user of web/mobile application	Internal analyst, for decision support
Machine use example	Checking if an action is authorized	Detecting fraud/abuse patterns
Type of queries	Fixed set of queries, predefined by application	Analyst can make arbitrary queries
Data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Data Warehouse Schema: Fact Tables

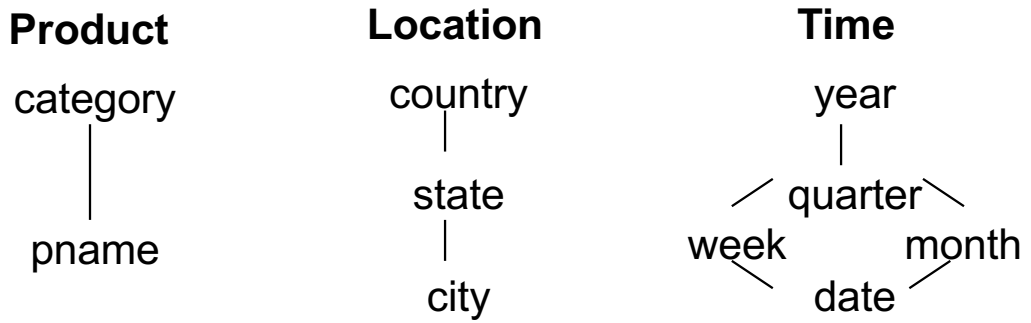
- Relational 'data warehouse' applications are centered around a **fact table**
  - For example, a supermarket application might be based on a table Sales (*Market_Id, Product_Id, Time_Id, Sales_Amt*)

market_id	product_id	time_id	sales_amt
M1	P1	T1	3000
M1	P2	T1	1000
M1	P3	T1	500
M2	P1	T1	100
M2	P2	T1	1100
M2	P3	…	…
…	…		

- The table can be viewed as multidimensional
  - Collection of numeric measures, which depend on a set of dimensions
    - E.g. Market_Id, Product_Id, Time_Id are the dimensions that represent specific supermarkets, products, and time intervals
    - Sales_Amt is a function of the other three

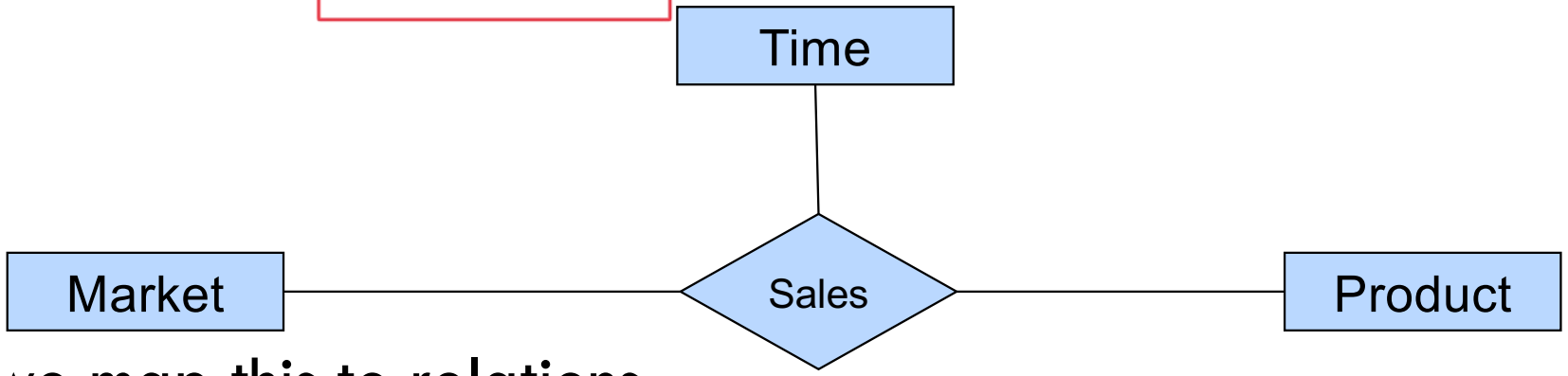
Data Warehouse Schema: Dimension Tables

- The dimensions of the fact table are further described with dimension tables
  - Supermarket Example: Fact table
    - Sales (*Market_id, Product_Id, Time_Id, Sales_Amt*)
  - Dimension Tables:
    - Market (*Market_Id, City, State, Region*)
    - Product (*Product_Id, Name, Category, Price*)
    - Time (*Time_Id, Week, Month, Quarter*)
- For each dimension, the set of values can be organized in a hierarchy:



Data Warehouse Schema: Star Schema

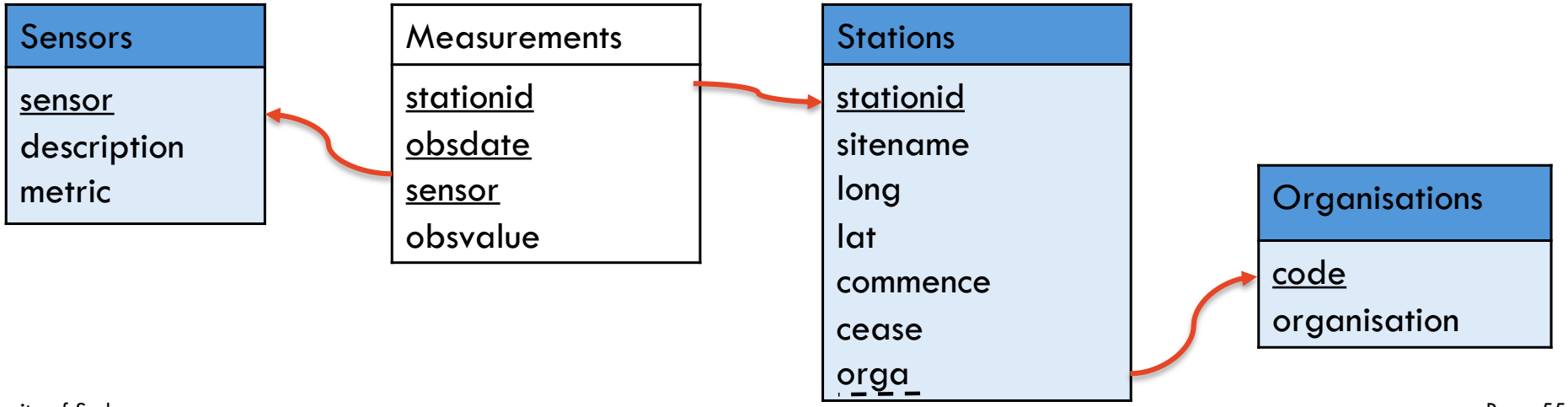
- The fact and dimension relations linked to it look like a star
- This is called a **star schema**



- If we map this to relations
  - 1 central fact table
  - n dimension tables with foreign key relationships from the fact table
    - the fact table holds the FKs referencing the dimension tables

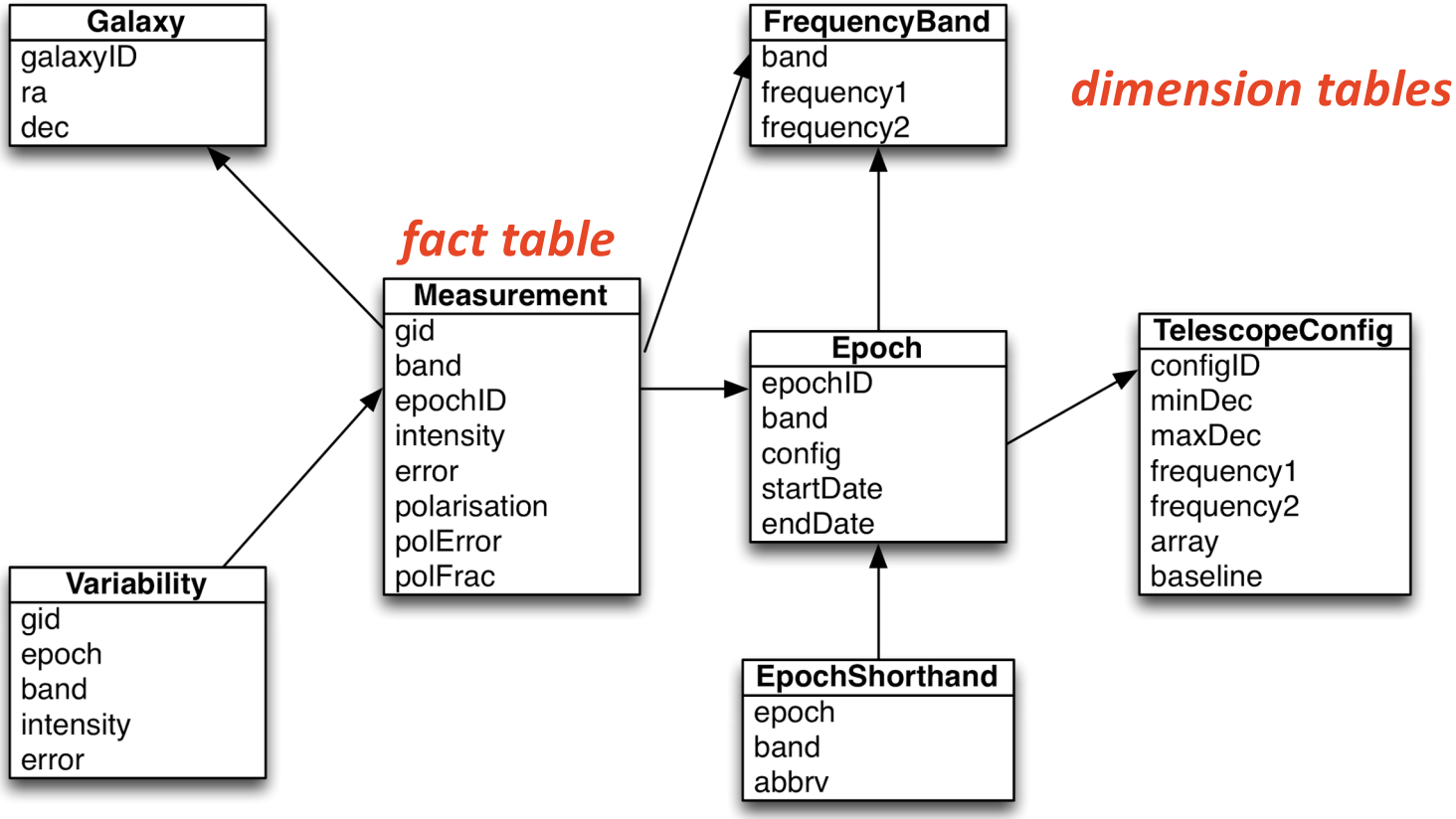
Example 1: 'WaterInfo' Star Schema

- Design Option 2 from WaterInfo use case: Normalised Relational Schema
  - measurements are **facts**, other data describes the **dimensions**
  - measurement entries get 'folded' into separate rows of the fact table
  - allows us to avoid NULLs as much as possible, but harder to interpret



Example 2: Astronomy Database Schema

Seven tables as shown below including foreign-key relationships

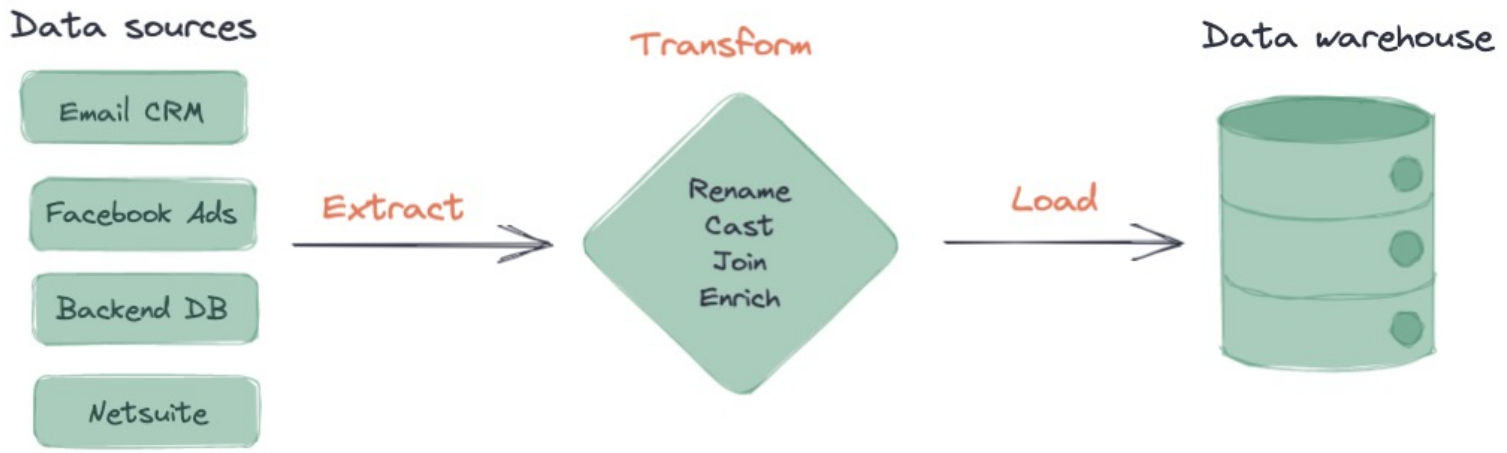


Summary

- Data warehousing principles relevant for data engineering too
  - ETL, Star Schema, OLAP queries, …
- Data warehouse can be used both as source, but more typically as the data storage layer for a data analytics pipeline
- Various systems available, especially cloud-based
  - Amazon Redshift
  - Snowflake
  - Google BigQuery
  - …

ETL vs ELT

Populating a Data Warehouse: ETL Process



https://docs.getdbt.com/terms/etl

Extract

- Purpose: Collect data from various sources.
- Examples:
  - Ad platforms (Facebook Ads, Google Ads)
  - Backend databases
  - Sales CRMs
- Methods:
  - Custom scripts using APIs
  - Open source/SaaS products
- Challenges:
  - High technical skill required
  - Maintenance of extraction scripts

Transform

- Purpose: Normalise and model raw data.
- Methods:
  - Custom Solutions:
    - Use programming languages like Python, Scala
    - Technologies like Apache Spark, Hadoop
  - ETL Products:
    - GUI-based platforms for low-code/no-code transformations
- Common Transformations:
  - Renaming columns
  - Casting fields correctly
  - Timestamp conversions

Load

- Purpose: Load transformed data into a storage layer, e.g., data warehouse
- Characteristics:
  - Only transformed data resides in the warehouse
  - Exposes data to BI tools and end users
- Challenges:
  - Ensure transformation accuracy without raw data in the warehouse

ETL Challenges

- Technical Limitations:
  - Limited version control
  - Excessive business logic in BI tools
  - Difficult QA processes

- Human Limitations:
  - Data analysts excluded from ETL processes
  - Lack of visibility for business users

Limited Version Control

- Transformations as standalone scripts or within ETL products are hard to version control.

- Lack of version control means data teams can't easily recreate or rollback historical transformations.

- Without version control, performing code reviews becomes challenging.

Immense amount of business logic living in BI tools

- Some teams with ETL workflows implement most of their business logic in their BI platform rather than earlier in the transformation phase.

- Most organizations have some business logic in their BI tools, but too much downstream logic can slow data rendering in the BI tool.

- Excessive downstream logic can be hard to track if the BI tool code is not version controlled or documented.

Data analysts are excluded from ETL work

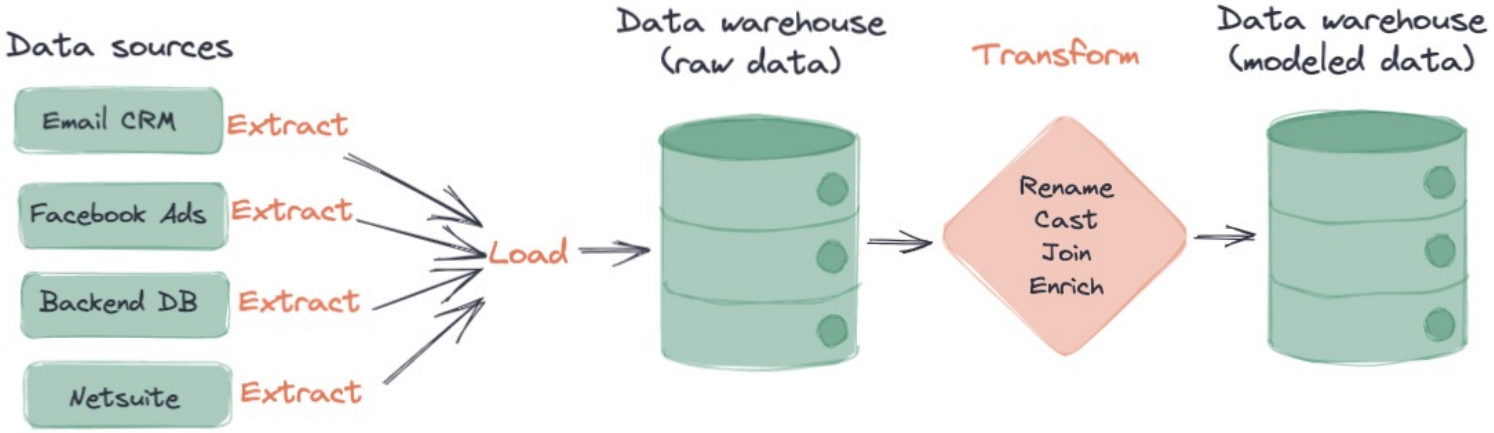
- ETL workflows often involve technical processes, restricting data analysts from participating in the data workflow.
- Data analysts' strength lies in their knowledge of data and SQL.
- When extractions and transformations use unfamiliar code or applications, data analysts' expertise may be excluded.
- Data analysts and scientists become dependent on others to create the necessary schemas, tables, and datasets.

Evolution from ETL to ELT

- Originally ETL was driven by resource limitations of both source and target systems, handling transformations external to both

- Evolution of ETL to ELT in two variants:
  - load extracted data into a *data lake* (e.g HDFS - Hadoop Distributed File System) and then transform for each use case
  - load extracted data into a data warehouse with minimal transformation, and then clean and transform it directly there

Populating a Data Warehouse: ELT Process



https://docs.getdbt.com/terms/etl

ELT - Load

- Purpose: Load extracted data into the target data lake/warehouse.

- Examples:
  - Snowflake
  - Amazon Redshift
  - Google BigQuery
  - Databricks Data Lakes

- Characteristics:
  - Data is mostly unchanged from extraction.
  - Light normalisation may occur in some scenarios.

ELT - Transform

- Purpose: Prepare raw data for analytics.
- Tasks:
  - Renaming columns
  - Correcting data types
  - Joining tables
  - Converting timestamps
  - **Unnesting JSON fields**
  - **Adding business logic**
  - **Establishing materialisations**
  - **Ensuring data quality (QA)**
- Methods:
  - Using dbt for transformations
  - Custom SQL scripts
  - Automated schedulers
  - Stored procedures

ELT vs ETL

- ELT:
  - Transformations occur after loading
  - Minimal coding for extraction and loading
  - Flexible and scalable modeling
  - Empowers data team members familiar with SQL

- ETL:
  - Transformations occur before loading
  - Requires custom scripts and technical skills
  - Encapsulated in one product
  - Heavy transformations often take place downstream in BI layer

Benefits of ELT

- Data as Code:
  - Version-controlled transformations
  - CI/CD workflows
  - Easy rollback of transformations
  - Scalable with business growth
- Empowerment:
  - Data analysts and scientists can create their own pipelines
  - Greater transparency and collaboration with business users
  - Documentation of transformation processes

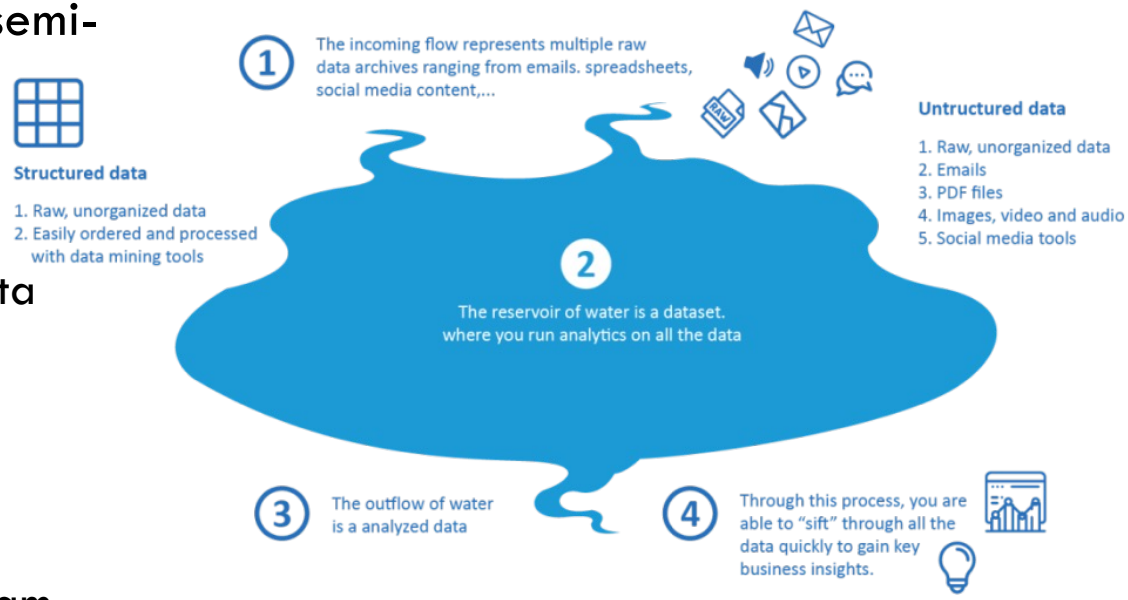
Data Lake

Evolution from Warehouse to Data Lake

- A data warehouse often uses a **relational** data model that is queried through SQL.
- This model works well for the types of queries that **business analysts** need to make.
- It is less well suited to the needs of **data scientists**, who might need to perform tasks such as:
  - Transform tabular data into a form that is suitable for training a machine learning model, e.g., feature engineering; this is difficult to express using SQL.
  - Take textual data (e.g., reviews of a product) and use natural language processing techniques to extract structured information from it (e.g., the sentiment of the author, or which topics they mention). Similarly, they might need to extract structured information from photos using computer vision techniques.

Data Lake

- A centralised storage repository that holds vast amounts of raw data in its native format, including structured, semi-structured, and unstructured data.
- Characteristics:
  - Can store large volumes of raw data without predefined schema
  - Data is loaded as-is, no need for preprocessing
  - Supports batch, real-time, and stream processing

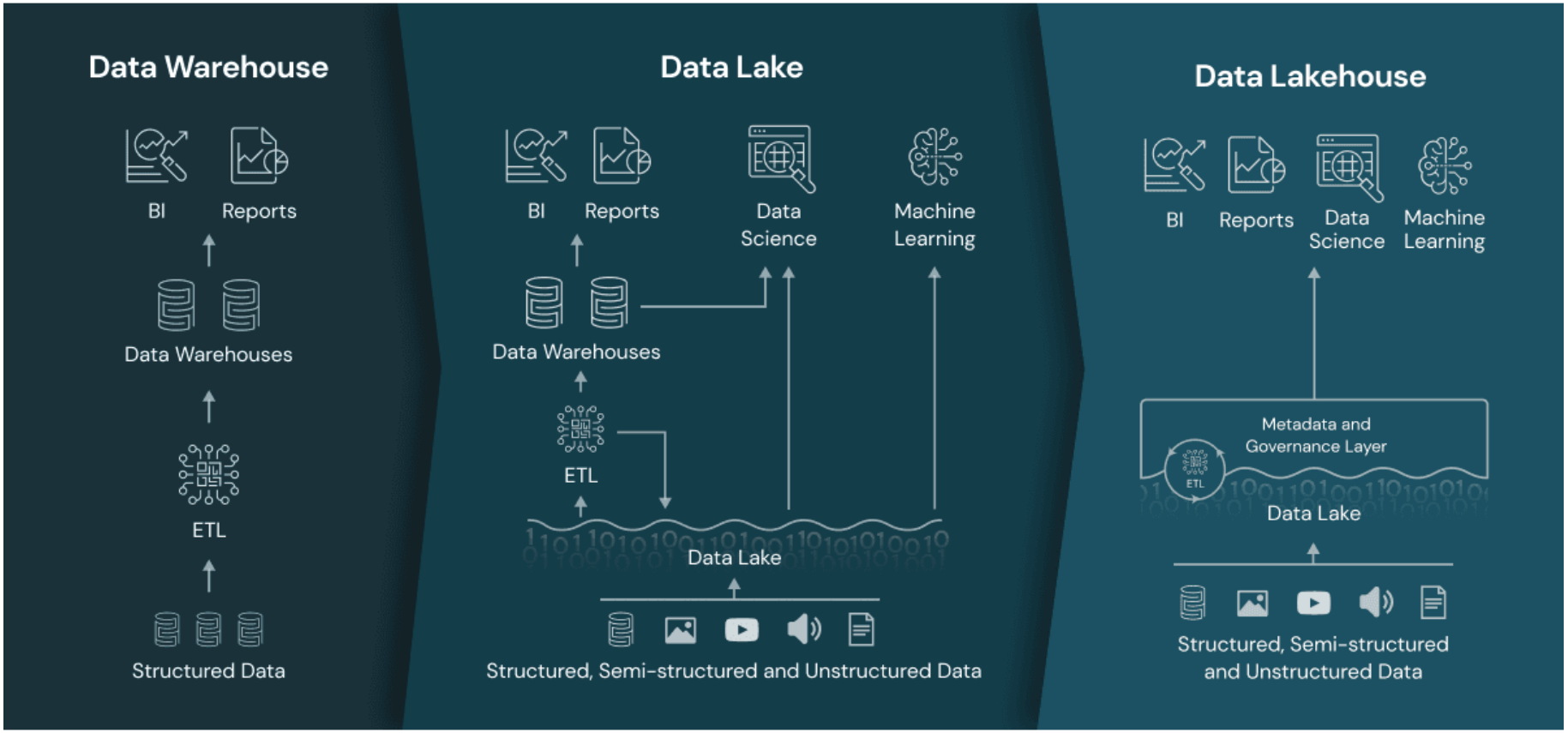


Data Lake vs Warehouse

	Schema	Data types	Agility	Processing	Cost
Lake	Schema-on-read (schema is defined at the time of reading data)	**Both structured and unstructured data**	More agile as it accepts raw data without a predefined structure	Extract – Load – Transform (ELT)	Cost-effective storage solutions, but costs can rise when processing large amounts of data
Warehouse	Schema-on-write (predefined schema before writing data)	**Primarily structured data**	Less agile due to predefined schema	Extract – Transform – Load (ETL)	Typically more expensive because of optimisations for complex queries

Data Lake vs Warehouse

- Use a Data Warehouse when:
  - You have structured data sources and require fast and complex queries.
  - Data integration from different sources is essential.
  - Business intelligence and analytics are the primary use cases.
- Use a Data Lake when:
  - You have a mix of structured, semi-structured, or unstructured data.
  - need a scalable and cost-effective solution to store massive amounts of data.
  - Future needs for data are uncertain, and you want flexibility in storage and processing.
- Often, organisations use a combination of both, ingesting raw data into a data lake and then processing and moving refined data into a data warehouse for analysis.



https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html