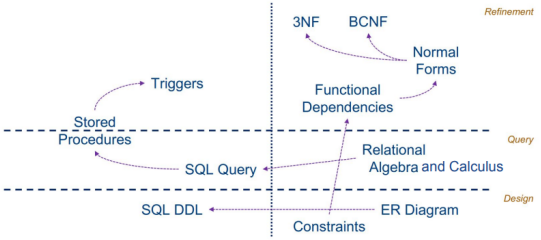
CS2102 Database Sys Summary

AY23/24 Sem 1, github.com/gerteck

Topics & Objectives

- **Design**: Entity-Relationship (ER) Model, Functional Dependencies, Normal Forms
- **Implementation**: SQL (Data definition language, Queries, Stored procedures, Triggers)
- **Theory**: Relational Calculus and algebra
- Module covers fundamental concpets and techniques for:
  - Understanding and practice of design & implementation of database applications and management of data with relational db management systems.
  - Design of ER data models to capture data requirements, translate to relational database schema, refine using schema decompositions to avoid anomalies.
  - Use SQL to define relational schemas, write queries.
  - Reason about correctness using concepts of formal query lang (relational calculus & algebra) and apply knowledge to develop database applications.



1. Database Management Sys DBMS

Challenges for Data-Intensive applications

- **Efficiency**: Fast access to information in volumes of data
- **Transactions**: "All or nothing" changes to data
- **Data Integrity**: Parallel access and changes to data
- **Recovery**: Fast and reliable handling of failures (e.g. HDD/Sys crash, power outage, network disruption)
- **Security**: Fine-grained data access rights

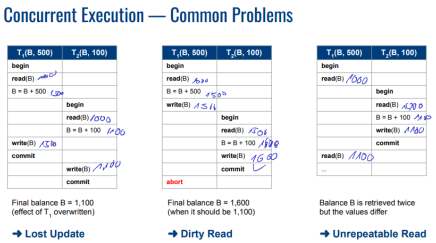
File-based data management to DBMS

- Complex, low level code, Often similar requirements across different programs
- **Problems**: High development effort, Long development times, Higher risk of (critical) errors
- **DBMS**: Set off universal and powerful functionalities for data management, with faster application development, higher stability, less errors.

Core concepts of DBMS

- **ACID Transaction**: Finite sequence of database operations (reads and/or writes), smallest logical unit of work
- **Atomicity**: either all effects of T are reflected in the database or none ("all or nothing")
- **Consistency**: the execution of T guarantees to yield a correct state of the database
- **Isolation**: execution of T is isolated from the effects of concurrent transactions
- **Durability**: after commit of T, its effects are permanent even in case of failures

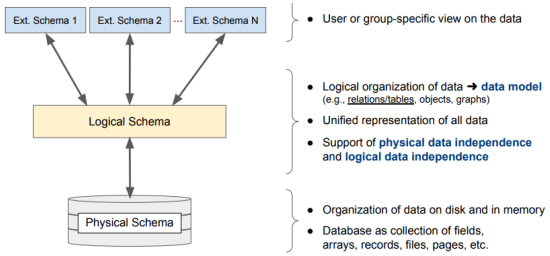
Concurrent Execution



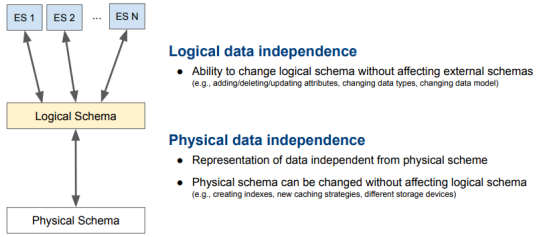
Require Serializable transaction execution:

- A concurrent execution of a set of transactions is serializable if this execution is equivalent to some serial execution of the same set of transactions
- Two executions are equivalent if they have the same effect on the data
- **DBMS**: Support concurrent executions of transactions to optimize performance, Enforce serializability of concurrent executions to ensure integrity of data

Data Abstraction



Data Independence



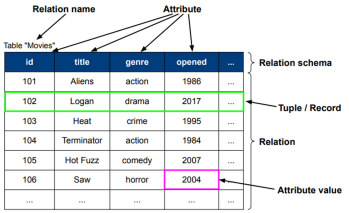
Terminology / Definitions

- **Data Model**: Collection of concepts for describing data
- **Schema**: Description of structure of DB using data model
- **Schema Instance**: Content of a DB at a particular time

Relational Data Model

Data is modelled by relations, and each relation has a definition called a relation schema. This schema specifies attributes (columns) and data constrains (e.g. domain constraints)

- **Relation**: Can be seen as Tables with rows and columns:
  - No. of cols = Degree/Arity, No. of rows = Cardinality
  - Each row is called a tuple/record. It has a component for each attribute of the relation.
  - A relation is thus a set of tuples and an instance of the relation schema, i.e. of a single table.
- **Domain**: Set of atomic values, e.g. integers. All values for an attribute is either in this domain or null.
- **Relational database schema**: Set of relation schemas and their data constraints, i.e. of multiple tables
- **Relational database**: Instance of the schema and is a collection of tables.



Term	Description (informal)
attribute	Column of a table
domain	Set of possible values for an attribute
attribute value	Element of a domain
relation schema	Set of attributes (with their data types + relation name)
relation	Set of tuples
tuple	Row of a table
database schema	Set of relation schemas
database	Set of relations / tables

Integrity Constraints

Condition that restricts the data that can be stored in a database instance. A legal relation instance is a relation that satisfies all specified ICs.

- **Domain Constraints**: Restrict the attribute values of relations, e.g. only integers allowed
- **Key Constraints**:
  - **Superkey**: A superkey is a subset of attributes in a relation that unique identifies its tuples.
  - **Key**: A key is a superkey which is minimal, i.e. no proper subset of itself is a superkey.
  - **Candidate keys**: Set of all possible keys for a relation. One of these keys is selected as the primary key.
  - **Primary key**: Chosen candidate key for a relation, Cannot be null (entity integrity constraint), Underlined in relation schema. Prime attribute: Attribute of a primary key (cannot be null)
- **Foreign Key Constraints**:
  - **Foreign key**: A foreign key refers to the primary key of a second relation (which can be itself)
  - Each foreign key value must be the primary key value in the referenced relation or be null (foreign key constraint)
  - Also known as referential integrity constraints.

Term	Description (informal)
(candidate) key	Minimal set of attributes that uniquely identify a tuple in a relation
primary key	Selected key (in case of multiple candidate keys)
foreign key	Set of attributes that is a key in referenced relation
prime attribute	Attribute of a (candidate) key

- Terminology: DB. vs DBS vs. DBMS

DBS = DBMS + n*DB (n>0)

2. SQL: Structured Query Language

- Declarative language: focus on what to compute, not on how to compute
- Contains two parts: Data Definition Language and Data Manipulation Language

Datatypes

type	description
boolean	logical Boolean (true/false)
integer	signed 4-byte integer
float8	double precision floating-point number (8 bytes)
numeric(p, s)	number with p significant digits and s decimal places
char(n)	fixed-length character string
varchar(n)	variable-length character string
text	variable-length character string
date	calendar date (year, month, day)
timestamp	date and time

Integrity Constraints 2

- A **consistent state** of the database is a state which complies with the with the business rules as defined by the structural constraints and the integrity constraints in the schema.
- If an integrity constraint is violated by an operation or a transaction, the operation or the transaction is aborted and rolled back and its changes are undone, otherwise, it is committed and its changes are effective for all users.
- Five main kinds of integrity constraints in SQL: **NOT NULL, PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK.**

Primary Key

A primary key is a set of columns that uniquely identifies a record in the table. Each table has at most one primary key. The primary key can be one column or a combination of columns.

```
-- Declare primary key as column constraint
CREATE TABLE customers (
 firstname VARCHAR(64 ,
 lastname VARCHAR(64) ,
 email VARCHAR(64) ,
id VARCHAR(16) PRIMARY KEY);
```

Composite Primary Key & NOT NULL

A not null constraint guarantees that no value of the column can be set to null. A not null constraint is always declared as a row constraint. When it is explicit, it is declared with the keyword NOT NULL.

```
CREATE TABLE games (
name VARCHAR(32) ,
version CHAR(3) ,
price NUMERIC NOT NULL,
PRIMARY KEY (name, version) );
```

Data Insertion Populating Tables

```
INSERT INTO customers VALUES(
'Carole', 'Yoga', 'cyoga@email.org',
'Carole89');
```

Deleting Tables

```
DELETE FROM customers
-- DROP deletes content \& table definition
DROP TABLE customers
DROP TABLE IF EXISTS downloads
```

Unique

A unique constraint on a column or a combination of columns guarantees the table cannot contain two records with the same value in the corresponding column or combination of columns.

```
CREATE TABLE customers (
firstname VARCHAR(64) NOT NULL ,
lastname VARCHAR(64) NOT NULL,
email VARCHAR(64) UNIQUE NOT NULL,
id VARCHAR(16) PRIMARY KEY,
UNIQUE (firstname, lastname));
```

Foreign Key

- A foreign key constraint enforces **referential integrity**. The values in the columns for which the constraint is declared must exists in the corresponding columns of the referenced table.

- Referenced columns are usually required to be the primary key of the referenced table. Some systems relax this.
- A foreign key is declared using the keyword REFERENCES as a row constraint and the keywords FOREIGN KEY and REFERENCES as a table constraint.

```
CREATE TABLE downloads (
customerid VARCHAR (16) REFERENCES
    customers (id),
name VARCHAR (32)
version CHAR (3),
FOREIGN KEY ( name, version) REFERENCES
    games (name, version) );
```

Check

- Check constraint enforces any other condition that can be expressed in SQL. Declared as row or table constraint.

```
CREATE TABLE games (
name VARCHAR(32),
version CHAR(3),
PRIMARY KEY (name, version)
price NUMERIC NOT NULL CHECK (price > 0)
-- or as table constraint:
CHECK (price > 0) );
```

Update and Delete Propagation

- The annotations ON UPDATE/DELETE with the option CASCADE propagate the update or deletion, when there are chains of foreign key dependencies.

```
CREATE TABLE downloads(
id VARCHAR (16) REFERENCES customers (id)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
name VARCHAR(32),
version CHAR(3),
PRIMARY KEY (id, name, version),
FOREIGN KEY (name, version) REFERENCES
    games(name, version)
  ON UPDATE CASCADE
  ON DELETE CASCADE);
```

- Generally a good idea to constraint all columns not to be null unless there is a good design or tuning reason for not doing so.
- Think carefully about which foreign keys should be subject to cascade.
- Good idea to defer all the constraints that can be deferred. These are checked at the end of a transaction and not immediately after each. operation.

Querying Tables

Print Table

- Wildcard '*' to include all attributes

```
SELECT *
FROM customers;
```

View

- We can give a name to a query, called a view. Once created, a view can be queried like a table.
- Creating a view is generally a better option than creating and populating a table, temporary or not.

```
CREATE VIEW sg\_customers AS
SELECT c.firstname, c.lastname, c.email,
    c.id
FROM customers c
WHERE country = 'Singapore';
```

```
SELECT * FROM sg\_customers;
```

3. SQL Queries

Printing one Table

```
SELECT firstname, lastname
FROM customers
WHERE country = 'Singapore';
```

DISTINCT and ORDER BY

- Selecting a subset of columns may result in duplicate row even if original table has a primary key.
- DISTINCT keyword eliminates eventual duplicates, requests results contain distinct rows.
- Both DISTINCT and ORDER BY involve sorting and conceptually ORDER BY is applied before SELECT DISTINCT.

```
SELECT DISTINCT name, version
FROM downloads
ORDER BY name ASC, version DESC;
```

WHERE

- Returns rows that evaluate to true, filter rows on a Boolean condition
- Uses Boolean operators such as AND, OR and NOT, and various comparison operators such as `>, <, >=, <=, <>, IN, LIKE` and `BETWEEN AND`
- Does not return rows that evaluate to unknown/null!
- '-' matches single char
- '%' matches any sequence of zero or more chars

```
SELECT firstname, lastname
FROM customers
WHERE country IN ('Singapore', 'Indonesia')
AND (dob BETWEEN '2001-01-01' AND
    '2000-12-01' OR since >= '2016-12-01 )
AND lastname LIKE 'B%'
```

- PostgreSQL use "——" for concatenation

```
-- Not to collect GST below 30 cents:
SELECT name || ' ' || version AS game,
    price * 1.07
FROM games
WHERE price * 0.07 < 0.3
```

De Morgan's Laws

```
SELECT name
FROM games
-- all 3 are the same:
WHERE (version = '1.0' or version = '1.1')
WHERE version IN ('1.0', '1.1')
WHERE NOT (version <> '1.0' AND version <>
    '1.1');
```

NULL value

• Every domain has additional value, null. Ambiguous, could be "unknown", "does not exists", or both. In SQL it is generally (but not always) "unknown".
• With null values, the logic of SQL is a three valued logic with unknown.
• Use `IS (NOT)NULL` for comparison with null
• `COALESCE()` returns the first non-null of its argument. • `COUNT(*)` counts NULL values. `COUNT(att)AVG(att)MAX(att)MIN(att)` eliminate null values

Cross Join

• Cross join & Cartesian Product & cross product, represented by comma `CROSS JOIN ,`
• Cross join with `WHERE` clause: add condition that FK columns equal to the corresponding PK columns.
• Systematically define table variables (e.g. `games AS g`)

```
SELECT *
FROM customers c, downloads d, games g
WHERE d.id = c.id
AND d.name = g.name
AND d.version = g.version
```

4. Algebraic SQL Queries

Inner Join

• `JOIN` interpreted as `INNER JOIN` • Inner joins combine records from two tables whenever there are matching values in a field common to both tables.

```
SELECT *
FROM customers c JOIN downloads d ON d.id =
    c.id
JOIN games g ON d.name = g.name AND
    d.version = g.version;
```

Natural Join

• If we give the same name to columns that are the same, can use natural join. Joins the rows that have the same values for their columns that have the same names. It also prints one of the two equated columns

```
SELECT *
FROM customers c NATURAL JOIN downloads d
    NATURAL JOIN games g;
```

Outer Join

• outer join keeps the columns of the rows in the left (left outer join), right (right outer join) or in both (full outer join) tables that do not match anything in the other table according to the join condition and pad the remaining columns with null values.
• Better to avoid outer joins whenever possible as they introduce null values.
• `RIGHT (OUTER)JOIN, LEFT (OUTER)JOIN`
• `FULL (OUTER)JOIN`

```
-- finds customers, never downloaded a game
SELECT c.id FROM customers c
LEFT JOIN downloads d ON c.id = d.id
WHERE d.id IS NULL;
```

Set Operations

• Union, intersect and non-symmetric difference.
• Eliminate duplicates: `UNION, INTERSECT, EXCEPT`
• Keep duplicates: `UNION ALL, INTERSECT ALL EXCEPT ALL`

4. Aggregate SQL Queries

Aggregate Functions

• The values of a column can be aggregated aggregation functions such as `COUNT(), SUM(), MAX(), MIN() AVG(), STDDEV()` etc.

```
SELECT COUNT(*)
FROM customers c;
```

```
-- ALL is default and omitted
-- DISTINCT needed
SELECT COUNT(ALL DISTINCT c.country)
FROM customers c;
```

```
-- Finds min, max, avg and stddev, TRUNC()
    displays 2 d p
SELECT MAX(g.price),
MIN(g.price),
TRUNC(AVG(g.price), 2) AS ave,
TRUNC(STDDEV(g.price), 2) AS std
FROM games g;
```

Group By

• The GROUP BY clause creates groups of records that have the same values for the specified fields before computing the aggregate functions.
• Groups are formed after the rows have been filtered by the WHERE clause
• Recommended (and required by SQL standard) to include attributes projected in the SELECT clause in the GROUP BY clause.
• The order of columns in the GROUP BY clause does not change the meaning of the query.

```
SELECT c.country, COUNT(*)
FROM customers c
WHERE c.dob >= '2000-01-01'
GROUP BY c.country
```

```
SELECT c.country, EXTRACT( YEAR FROM
    c.since) AS regyear, COUNT(*) AS total
FROM customers c, downloads d
WHERE c.id = d.id
GROUP BY c.country, regyear,
ORDERBY regyear, c.country;
```

Having

• Aggregate functions can be used in conditions. However, agg. functions not allowed in WHERE.
• `HAVING` clause to add conditions to be checked after the evaluation of the GROUP BY.
• `HAVING` can only involve aggregate functions, columns listed in the GROUP BY clause and subqueries.

```
SELECT c.country,
FROM customers c
GROUP BY c.country
HAVING COUNT(*) >= 100;
```

5. Nested SQL Queries

Subqueries

• In FROM clause: Must be enclosed in parenthesis, Table alias mandatory, Column aliases optional
• Not recommended, can be written as simple query

```
SELECT cs.lastname, d.name
FROM (SELECT *
   FROM customers c
  WHERE c.country = 'Singapore') AS cs,
      downloads d
WHERE cs.id = d.id;
```

• In WHERE clause, also can be written as simple query.
• Never use a comparison to a subquery without specifiying the quantifier ALL or ANY

```
SELECT g1.name, g1.version, g1.price
FROM games g1
WHERE g1.price >= ALL (
  SELECT g2.price
  FROM games g2);
-- or do:
WHERE g1.price = ALL (
  SELECT MAX(g2.price)
  FROM games g2);
-- Note HAVING g.price=MAX(g.price) will
    not work
```

Exists

• `EXISTS` evaluates to true if the subquery has some results.
• Generally correlated. If uncorrelated, then likely either wrong or unnecessary

```
SELECT c.id
FROM customers c
WHERE NOT EXISTS (
  SELECT d.id
  FROM downloads d
  WHERE c.id = d.id);
-- same as
WHERE c.id NOT IN (
  SELECT d.id
  FROM downloads d);
-- same as
WHERE c.id <> ALL (
  SELECT d.id
  FROM downloads d);
```

```
-- Find countries with most customers
SELECT c1.country
FROM customers c1
GROUP BY c1.country
HAVING COUNT(*) >= ALL (
  SELECT COUNT(*)
  FROM customers c2
  GROUP BY c2.country);
```

SQL Conditionals

CASE expression • Generic conditional, like if/else statement, Used in SELECT, ORDER BY, etc

```
-- Regular if else
CASE
  WHEN cond1 then res1
  WHEN cond2 then res2
  ...
  WHEN condN then resN
  ELSE res0
END
```

```
-- Switch like statement
CASE expression
  WHEN val1 then res1
  WHEN val2 then res2
  ...
  WHEN valN then resN
  ELSE res0
END
```

Conceptual evaluation of queries

FROM → WHERE → GROUP BY → HAVING → SELECT → ORDER BY → LIMIT/OFFSET