SQL essentials

Mastering database queries:
a comprehensive guide to SQL

Learning session 08

Query execution order. Indexes.
Understanding transactions
and ACID properties.

Instructor
Péter Fülöp
peteristvan.fulop@hcl.com

Query execution order

The query execution order (part 1)

1

FROM/JOIN Clause
The database engine starts by looking at the **FROM** clause. This is where it determines which tables to scan. Then the **JOIN ... ON** clause is evaluated to determine how to merge rows from the joined tables.

2

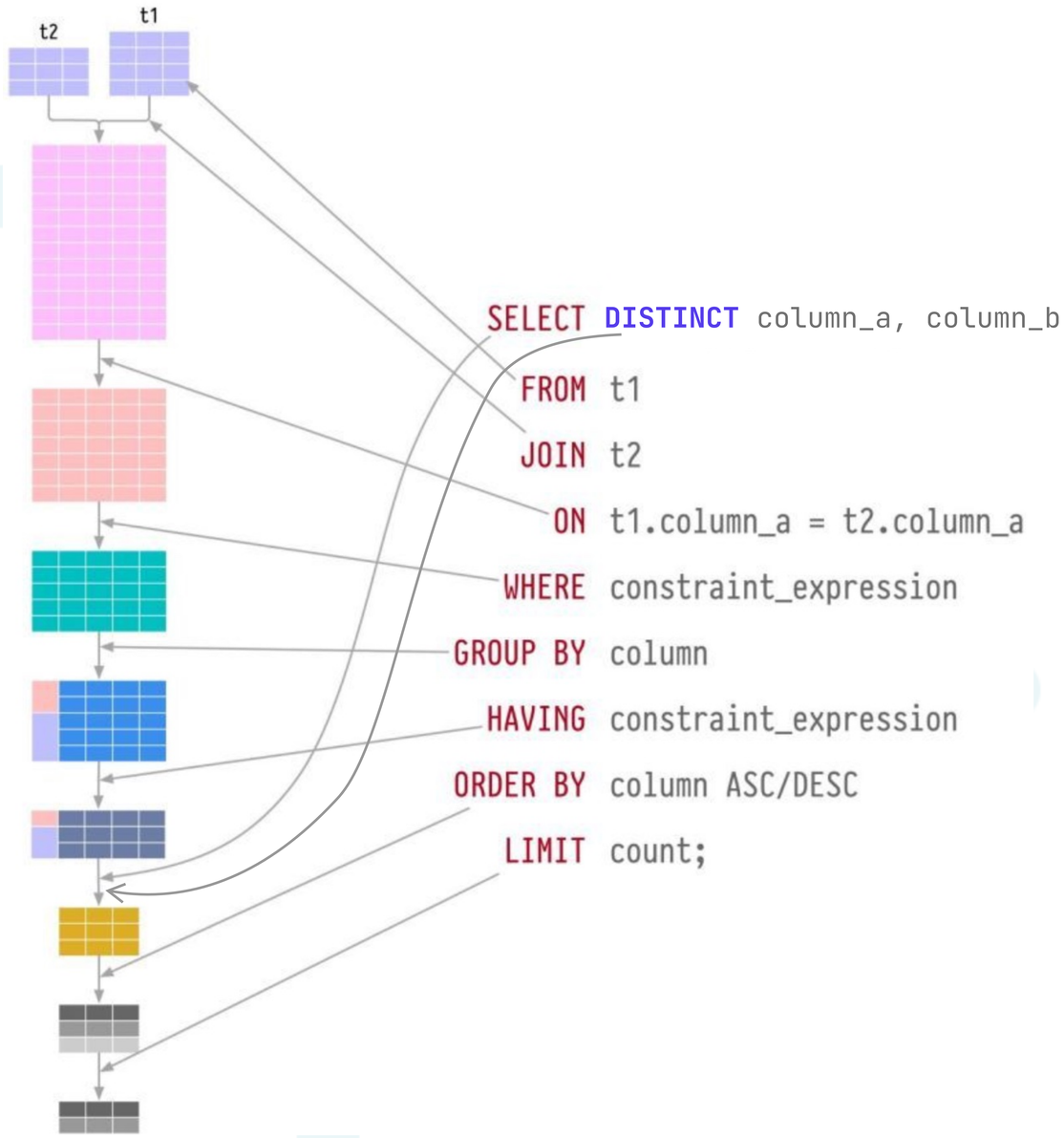
WHERE Clause
Filters the rows according to the conditions specified. This is done before grouping or selecting columns.

3

GROUP BY Clause
Rows are then grouped based on the specified columns. Essential for aggregate functions.

4

HAVING Clause
Groups are filtered based on the HAVING condition.



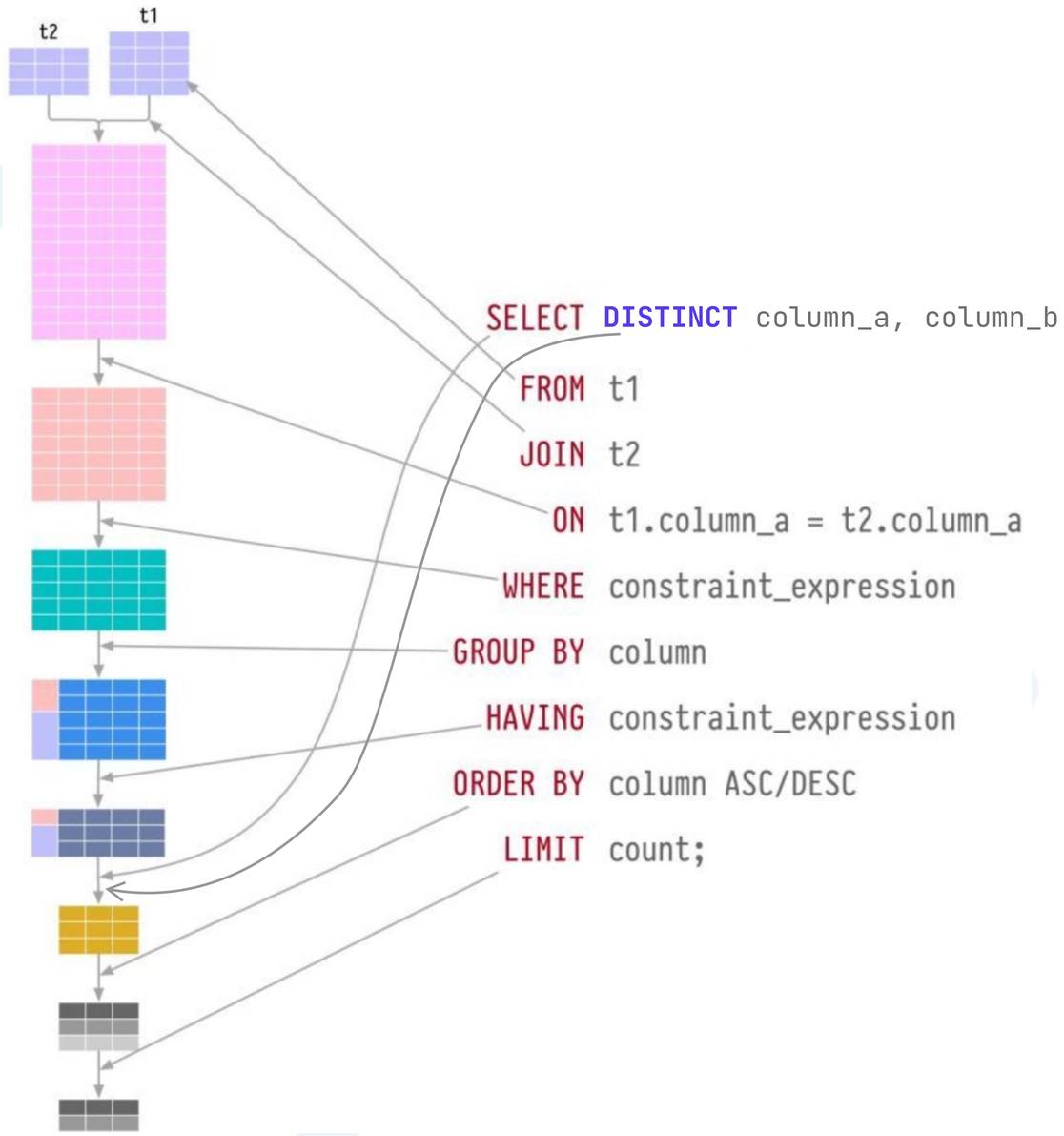
The query execution order (part 2)

5 **SELECT Clause**
Selection of columns and computation of expressions.

6 **DISTINCT**
Sorting of the result set.

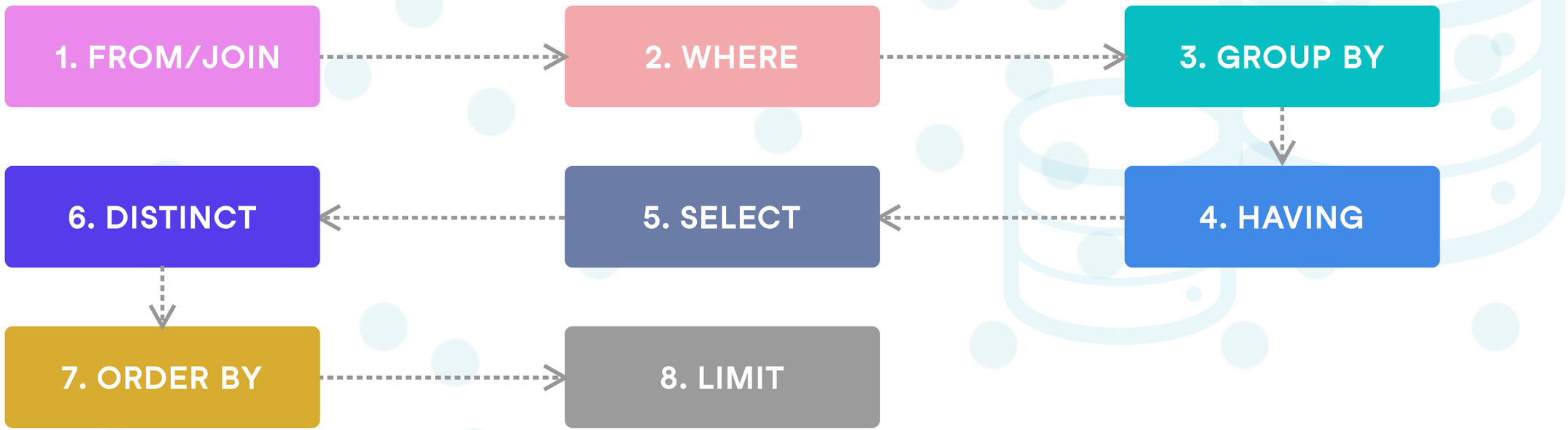
7 **ORDER BY Clause**
Sorting of the result set.

8 **LIMIT/OFFSET Clause**
Finally, a subset of rows is returned based on LIMIT and OFFSET.



The query execution order (summary)

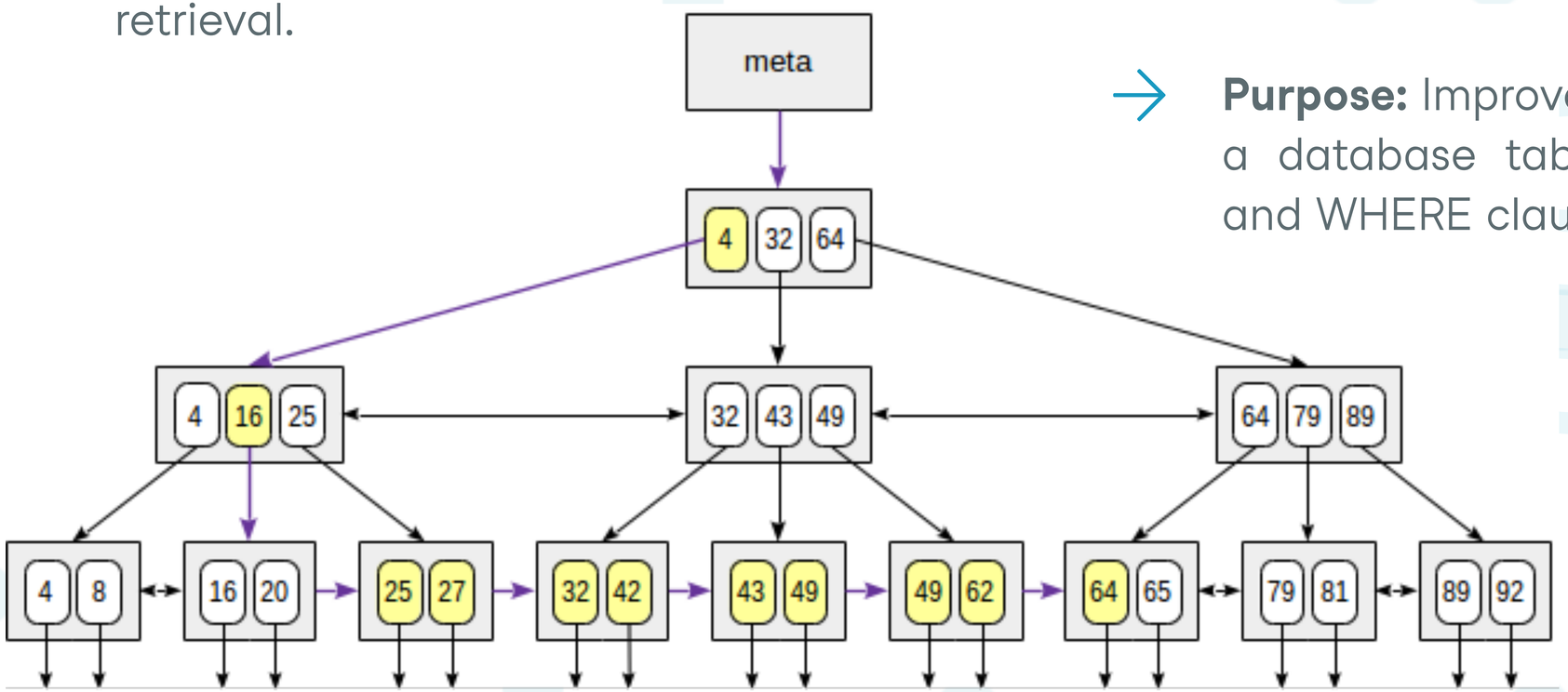
Understanding the execution order is crucial for query optimization and prediction.



Introduction to indexes

What are Indexes?

- **Definition:** Indexes are special lookup tables/objects that the database search engine can use to speed up data retrieval.
- **Analogy:** Like a book's index, they help in finding information quickly without scanning every page.
- **Purpose:** Improve the speed of operations in a database table, particularly for SELECT and WHERE clauses.



Why are Indexes Important?

Performance Enhancement: Significantly reduce the time to retrieve data from a database.

Efficiency in Large Databases: Essential for improving performance in tables with large volumes of data.

Trade-off: While they speed up data retrieval, they can slow down data insertion, deletion, and updating, due to the need to maintain the index.

Types of Indexes

Single-Column Indexes: Created on a single column of a table. Useful for queries that frequently search or sort based on that column.

Composite Indexes: Involve multiple columns. Useful for queries that filter or sort on multiple columns.

Unique Indexes: Ensure that two rows of a table do not have duplicate values in a specific column or combination of columns

Full-Text Indexes: Designed for text-based columns. They allow fast searching for words within text columns.

How Indexes Work

Structure / Implementation: different implementations of an index will improve query performance for different type of operators:

1. **B-Tree Index** - very useful for single value search or to scan a range, but also for pattern matching.
2. **Hash Index** - very efficient when querying for equality.
3. Generalized Inverted Index (GIN) - useful for indexing array values and testing the presence of an item.
4. Block Range Index (BRIN) - this type of index stores summary information for each table block range
5. Special indexes like GiST, SP-GiST and PostgreSQL comes with an extension to define custom index types

Process: When a query is executed, the database engine first checks if an index can be used to speed up the process.

Selection: The database's query optimizer decides whether to use an index, based on the query and the index's structure.

Introduction to indexes

Considerations for Using Indexes

Not Always Beneficial: Indexes are not suitable for tables with frequent, large batch updates or inserts

Maintenance:
They require additional storage and maintenance overhead.

Balance: Effective use of indexes involves balancing between the speed of read operations and the overhead of write operations.

Best practices

Index Key Columns: Choose columns that are frequently used in query conditions.

Monitor Performance: Regularly monitor the performance impact of indexes.

Avoid Over-Indexing: Too many indexes can degrade performance, especially in write-heavy databases.

Sargable queries

A sargable query, short for "**Search ARGument ABLE**," is a query that can take advantage of indexes effectively to optimize the search process.

SQL

```sql
--bad
SELECT * FROM employees WHERE EXTRACT(YEAR FROM hire_date) > 2020;
--good
SELECT * FROM employees WHERE hire_date > TO_DATE('2020-12-31', 'YYYY-MM-DD') ;

--bad
SELECT * FROM employees WHERE SUBSTRING(last_name from 1 for 1)='K;
--good
SELECT * FROM employees WHERE last_name LIKE 'K%';
```

Introduction to transactions

Introduction to Transactions

Definition: A transaction in SQL is **a sequence of one or more SQL operations** that are treated as a single unit. This unit must either be completed in its entirety or not executed at all.

Importance: Transactions ensure data integrity and consistency in database operations.

SQL

```sql
BEGIN; -- or START TRANSACTION
-- This command initiates a new transaction. It's the point at which PostgreSQL
-- starts to keep track of the changes you want to make.
-- Within a transaction, you can execute multiple SQL commands like SELECT,
-- INSERT, UPDATE, DELETE, etc. These operations are part of the transaction
-- and will either all be committed or rolled back together.
COMMIT; -- This command is used to save all changes made during the current transaction.
-- Once a transaction is committed, the changes are permanent and visible to other users.
```

Example 1: using transaction with COMMIT

Suppose you have a `bank_accounts` table in your PostgreSQL database with the following columns: account_id, account_holder, and balance. You want to transfer 100$ from account 101 to account 102.
This operation involves two main steps: (1) debiting one account and (2) crediting another.
Both steps need to be executed successfully to complete the transaction.

SQL

```sql
BEGIN;
UPDATE bank_accounts SET balance = balance - 100 WHERE account_id = 101;
UPDATE bank_accounts SET balance = balance + 100 WHERE account_id = 102;
COMMIT;
```

(1) The **BEGIN** command starts the transaction.
(2) The **first UPDATE** statement deducts $100 from the sender's account (account_id 101).
(3) The **second UPDATE** statement adds $100 to the receiver's account (account_id 102).
(4) The **COMMIT** command is used to save the changes made by both update statements.
(5) If either of the UPDATE statements fails (for example, due to a constraint violation, nonexistent account_id, etc.), PostgreSQL will raise an error. When an error is raised, the transaction is left in an uncommitted state. **It's important to note that PostgreSQL does not automatically roll back the transaction in this case.** The transaction remains 'open', and no changes are committed yet.

Example 1: rolling back the failling transaction

Anonymous DO block SQL

```sql
DO $$
BEGIN
    -- Start the transaction
    BEGIN;

    -- Perform the update operations
    UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

    -- Commit the transaction
    COMMIT;

EXCEPTION
    WHEN OTHERS THEN
        -- In case of any error, rollback the transaction
        ROLLBACK;
        RAISE; -- Optional: re-raise the exception for logging or further handling
END;
$$;
```

Example 2: using transaction with ROLLBACK

Imagine you are managing a database for a bookstore. You have two tables: books and orders.
The books table tracks the inventory of books, and the orders table records customer orders. You want to perform an operation where you update the inventory in the books table and add a new order to the orders table.
However, you need to ensure that both operations succeed or fail together to maintain data consistency.

SQL

```sql
BEGIN;

UPDATE books SET quantity = quantity - 2 WHERE id = 1;
INSERT INTO orders (book_id, customer_name, quantity) VALUES (1, 'John Doe', 2);

-- Simulating an issue
ROLLBACK;

-- Verify the rollback
SELECT * FROM books WHERE id = 1;
SELECT * FROM orders WHERE book_id = 1 AND customer_name = 'John Doe';
```

Key Properties of Transactions (ACID)

A

Atomicity
Ensures that all operations within a transaction are completed successfully or none are applied.of columns and computation of expressions.

C

Consistency
Guarantees that a transaction transforms the database from one valid state to another

I

Isolation
Ensures that transactions are securely and independently processed, maintaining data integrity.

D

Durability
Once a transaction is committed, it will remain so, even in the event of system failures.

Concurrent Transactions (Part 1)

Concurrent transactions refer to multiple transactions that are executed at the same time.
In a busy database system, it's common for several transactions to overlap in time. For example, while one transaction is reading data from a table, another might be updating or deleting data from the same table.

The ANSI/ISO SQL standard defines four levels of transaction isolation: uncommitted read, committed read (the default isolation level in PostgreSQL), repeatable read, and serializable.

The phenomena which are prohibited at various levels are:

1. **dirty read**
   A transaction reads data written by a concurrent uncommitted transaction.

2. **nonrepeatable read**
   A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

Concurrent Transactions (Part 2)

The phenomena which are prohibited at various levels are:

3

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

4

serialization anomaly

A serialization anomaly occurs when the outcome of a set of transactions is inconsistent with any possible order in which those transactions might have been executed serially (i.e., one after the other, without overlapping).

SQL

```sql
-- Setting the isolation level to Serializable
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- Your transaction operations here
COMMIT;
```

https://www.postgresql.org/docs/current/transaction-iso.html

https://www.youtube.com/watch?v=G8wDjVON9tk

Creating tables in SQL

Types of Indexes

A relational database consists of multiple related tables. Each **table is structured into rows and columns**, providing a framework for organizing structured data such as customer details, employee records, product information, and more.

SQL

```
CREATE TABLE [IF NOT EXISTS] table_name (
    column1 datatype(length) column_contraint,
    column2 datatype(length) column_contraint,
    column3 datatype(length) column_contraint,
...
    table_constraints
);
```

- Firstly, after the **CREATE TABLE** keywords, you need to define the name of your new table.
- Secondly, if you attempt to create a table that already exists, it will lead to an error. To avoid this, you can use the **IF NOT EXISTS** option.
- Thirdly, you should **list the columns of the table**, separated by commas. Each column is defined by its name, the type of data it holds, the data length, and any column constraints. These constraints are rules that the data in the column must adhere to.
- Finally, you need to define any **table-level constraints**, such as primary key, foreign key, and check constraints.

Example 1: creating tables

SQL

```sql
CREATE TABLE job_titles
(
    id     serial PRIMARY KEY,
    name   text NOT NULL,
);
```

The code snippet creates a new table named business_units with three columns: id, name, and activity_start_date.

1. `CREATE TABLE job_titles:` This line starts the command to create a new table named job_titles.

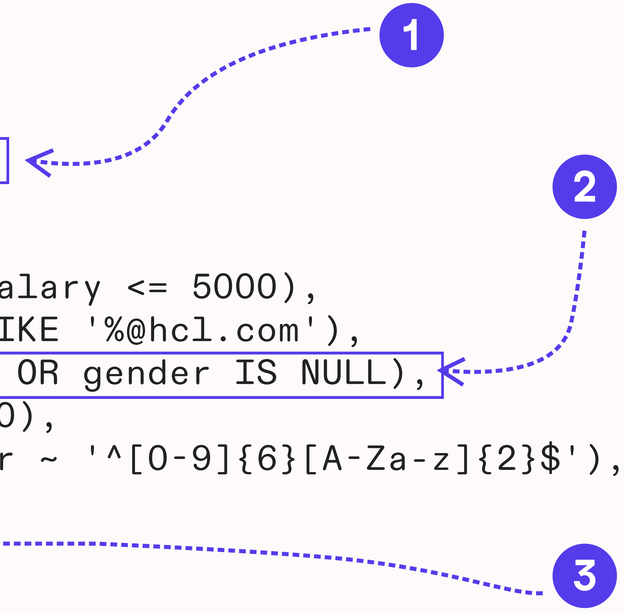
2. `id serial PRIMARY KEY:` the serial data type is used for auto-incrementing integer columns. It's commonly used for primary keys.

3. `name text NOT NULL:` the text data type is used for string values with variable length. It can store strings with a practically unlimited length.
`NOT NULL:` This constraint ensures that this column cannot have NULL values. Every record in the table must have a value for name.

Example 2: creating tables

SQL

```sql
CREATE TABLE employees (
    id                          SERIAL PRIMARY KEY,
    first_name                  VARCHAR(25) NOT NULL,
    middle_name                 VARCHAR(25),
    last_name                   VARCHAR(25) NOT NULL,
    hire_date                   DATE        NOT NULL,
    birth_date                  DATE        NOT NULL,
    business_unit_id            INTEGER,
    lead_employee_id            INTEGER REFERENCES employees (id),
    job_title_id                INTEGER REFERENCES job_titles (id),
    band_id                     INTEGER REFERENCES bands (id),
    last_salary_review_date     DATE,
    salary                      INTEGER CHECK (salary >= 1500 AND salary <= 5000),
    email_address               VARCHAR(255) CHECK (email_address LIKE '%@hcl.com'),
    gender                      CHAR(1) CHECK (gender IN ('M', 'F') OR gender IS NULL),
    yearly_leave_days           INTEGER CHECK (yearly_leave_days > 0),
    personal_id_number          VARCHAR(8) CHECK (personal_id_number ~ '^[0-9]{6}[A-Za-z]{2}$'),
    date_of_last_medical_analysis DATE,
    residence_country_id        VARCHAR     NOT NULL,
    native_language_id          INTEGER REFERENCES languages (id),
    preferred_language_id       INTEGER REFERENCES languages (id) );
```



Example 2: creating tables

1

`job_title_id INTEGER REFERENCES job_titles (id),`

`job_title_id:` This is a column in the employees table.

INTEGER: This data type indicates that the column will store integer values.

REFERENCES job_titles (id): this is a foreign key constraint.
It establishes a relationship between this column and the id column in the job_titles table.
This means that every value in the job_title_id column must exist as a value in the id column of the job_titles table.
It's used to ensure **referential integrity between the two tables**, meaning you can't have a job title ID in the employees table that doesn't exist in the job_titles table.

2

`CHECK (gender IN ('M', 'F') OR gender IS NULL):` This is a check constraint on the gender column. It ensures that the only values that can be stored in this column are 'M' for male, 'F' for female, or a NULL value.

3

`residence_country_id VARCHAR NOT NULL:` this constraint means that every record in the employees table must have a value for residence_country_id. It cannot be left empty (NULL), ensuring that the country of residence is always recorded.

Modify an existing table structure

SQL

```sql
ALTER TABLE table_name
[operation] [column_name] [new specifications];
```

SQL

```sql
-- adding a new column
ALTER TABLE employees
ADD COLUMN phone_number VARCHAR(15);
```

SQL

```sql
-- modify a column data type
ALTER TABLE employees
ALTER COLUMN salary TYPE NUMERIC(10, 2);
```

SQL

```sql
-- drop a column
ALTER TABLE employees
DROP COLUMN personal_id_number;
```

SQL

```sql
-- change a column name
ALTER TABLE employees
RENAME COLUMN birth_date TO date_of_birth;
```

What's next?

Opportunities

1

Capstone project

Demonstrate the skills and knowledge you've acquired throughout the course by undertaking and completing the capstone project.

2

Intermediate SQL: building on fundamentals

We are currently in the planning stages for our next SQL course, which will introduce advanced topics such as window functions, transaction management, anonymous code blocks, and more. Stay tuned for further details.

3

Community event

Our mentoring team is organizing a special community event titled "Automatizing and Visualizing Your Database Queries," specifically tailored for you.

4

Practice

Keep practicing consistently. Opt for online tests (like those available on Datacamp) to enhance and solidify your understanding.