Database Indexes, Triggers and Transactions

Databases and Web Applications Laboratory (LBAW)
Bachelor in Informatics Engineering and Computation (L.EIC)

Sérgio Nunes
Dept. Informatics Engineering
FEUP · U.Porto

Outline

- ➔ Database Specification (EBD) development (A6)
  - ➔ Indexes
  - ➔ Triggers
  - ➔ Transactions
  - ➔ Database Population

- ➔ PostgreSQL setup

LBAW Plan

- ➔ Plan: *Moodle Lecture #4*
  - ➔ 4th week of classes;
  - ➔ Continue development on the second component (EBD);
  - ➔ Lab classes:
    - ➔ work on component (EBD);
    - ➔ work on the conceptual data model (A4); [**important**]
    - ➔ work on the relational schema (A5).
- ➔ Monitor sessions: Tuesdays, from 16h to 17h30
  - ➔ PostgreSQL setup and use.

Database Specification (EBD) Development

Database Specification (EBD) Component

review

- The EBD component groups the artifacts to be made by the development team in order to support the storage and retrieval requirements identified in the requirements specification.
- It consists of three artifacts:
  - A4: Conceptual Data Model
  - A5: Relational Schema, Validation and Schema Refinement
  - A6: Indexes, Triggers, Transactions and Database Population

A4. Conceptual Data Model

review

➔ In this artifact the data requirements of the system are detailed.

➔ The Conceptual **Domain** Model contains the identification and description of the entities of the domain and the relationships between them.

➔ The Conceptual Domain Model is simplified to include only concepts (entities and relationships) of the domain that are stored in the database.

➔ The Conceptual **Data** Model is obtained by using a UML class diagram containing the classes, associations, multiplicity and roles.

➔ For each class, the attributes, associations and constraints are included in the class diagram.

➔ Business rules not included in the class diagram are described by words or using OCL (Object Constraint Language) included as UML notes.

A5. Relational Schema, Validation and Schema Refinement

review

- ➔ The A5 artifact contains the Relational Schema obtained by mapping from the Conceptual Data Model.
- ➔ The Relational Schema includes each relation schema, attributes, domains, primary keys, foreign keys and other integrity rules: UNIQUE, DEFAULT, NOT NULL, CHECK.
- ➔ Relation schemas are specified in the compact notation.
- ➔ In addition to this representation, the relational schema is also presented in SQL as an annex.
- ➔ To validate the Relational Schema obtained from the Conceptual Model, all functional dependencies are identified and the normalization of all relation schemas is accomplished.
- ➔ Should it be necessary, in case the scheme is not in the Boyce–Codd Normal Form (BCNF), the relational schema is refined using normalization.

A5. Relational Schema Compact Notation

review

- Relation schemas are specified in the compact notation:

- table1(id, attribute NN)
  table2(id, attribute → Table1 NN)
  table3(id1, id2 → Table2, attribute UK NN)
  table4((id1, id2) → Table3, id3, attribute)

- Primary keys are underlined. UK means UNIQUE and NN means NOT NULL.

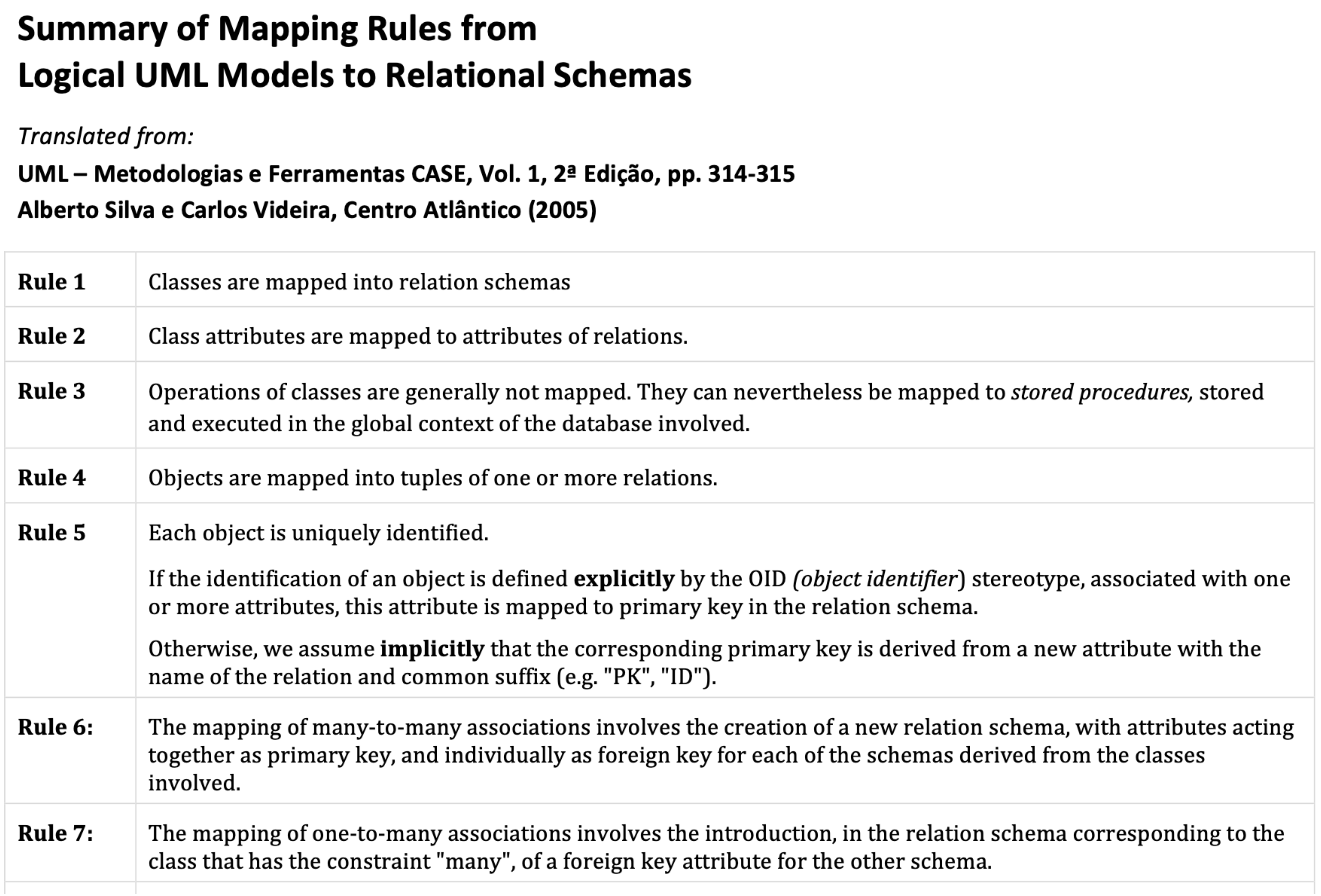
- The specification of additional domains can also be made in a compact form, using the notation:

- Today DATE DEFAULT CURRENT_DATE
  Priority ENUM ('High', 'Medium', 'Low')

- **In PostgreSQL use lower case and the "snake_case" convention.**

A5. Relational Schema Mapping

review



Summary of Mapping Rules from
Logical UML Models to Relational Schemas

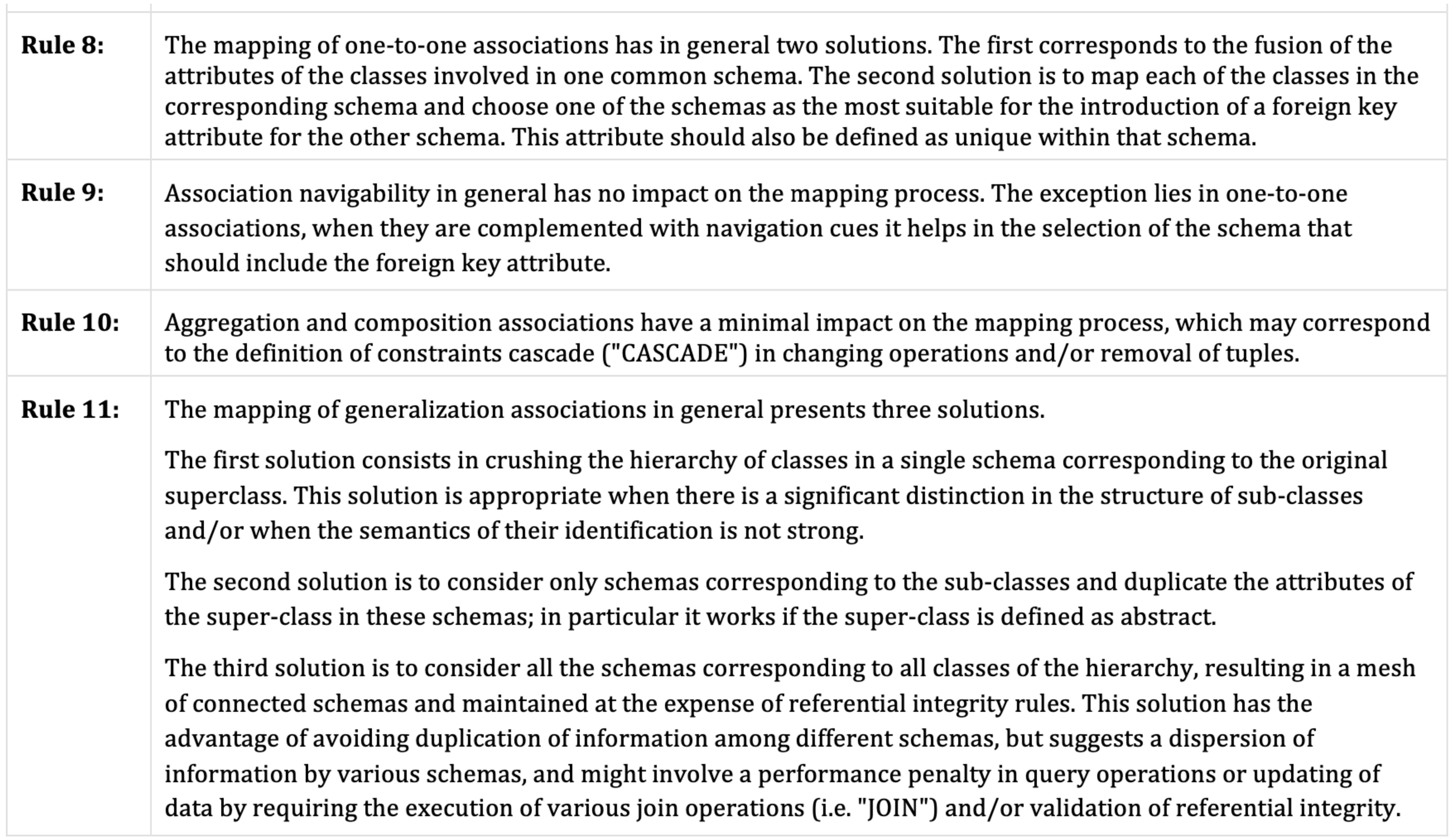
Translated from:
UML – Metodologias e Ferramentas CASE, Vol. 1, 2ª Edição, pp. 314-315
Alberto Silva e Carlos Videira, Centro Atlântico (2005)

Rule 1	Classes are mapped into relation schemas
Rule 2	Class attributes are mapped to attributes of relations.
Rule 3	Operations of classes are generally not mapped. They can nevertheless be mapped to *stored procedures,* stored and executed in the global context of the database involved.
Rule 4	Objects are mapped into tuples of one or more relations.
Rule 5	Each object is uniquely identified. If the identification of an object is defined **explicitly** by the OID *(object identifier*) stereotype, associated with one or more attributes, this attribute is mapped to primary key in the relation schema. Otherwise, we assume **implicitly** that the corresponding primary key is derived from a new attribute with the name of the relation and common suffix (e.g. "PK", "ID").
Rule 6:	The mapping of many-to-many associations involves the creation of a new relation schema, with attributes acting together as primary key, and individually as foreign key for each of the schemas derived from the classes involved.
Rule 7:	The mapping of one-to-many associations involves the introduction, in the relation schema corresponding to the class that has the constraint "many", of a foreign key attribute for the other schema.

Alberto Manuel Rodrigues da Silva e Carlos Alberto Escaleira Videira, UML, metodologias e ferramentas CASE, 2ª Edição, Volume 1, Centro Atlântico Editora, Maio 2005.

A5. Relational Schema Mapping

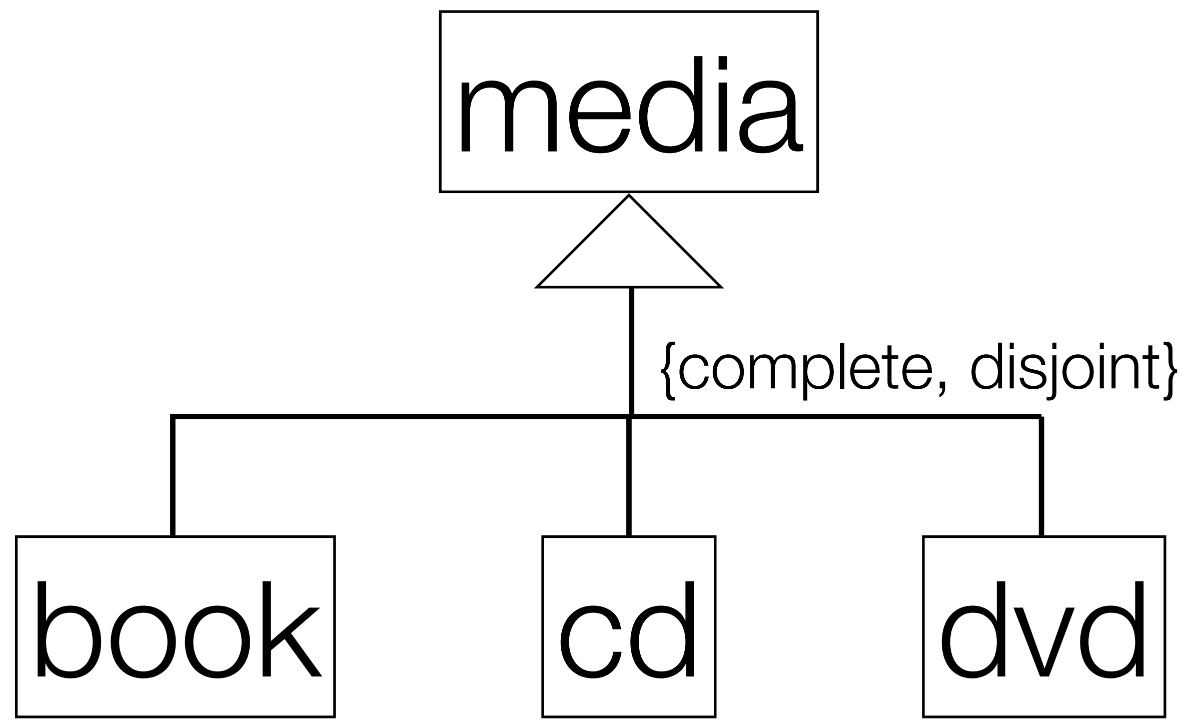
review



Rule 8:	The mapping of one-to-one associations has in general two solutions. The first corresponds to the fusion of the attributes of the classes involved in one common schema. The second solution is to map each of the classes in the corresponding schema and choose one of the schemas as the most suitable for the introduction of a foreign key attribute for the other schema. This attribute should also be defined as unique within that schema.
Rule 9:	Association navigability in general has no impact on the mapping process. The exception lies in one-to-one associations, when they are complemented with navigation cues it helps in the selection of the schema that should include the foreign key attribute.
Rule 10:	Aggregation and composition associations have a minimal impact on the mapping process, which may correspond to the definition of constraints cascade ("CASCADE") in changing operations and/or removal of tuples.
Rule 11:	The mapping of generalization associations in general presents three solutions. The first solution consists in crushing the hierarchy of classes in a single schema corresponding to the original superclass. This solution is appropriate when there is a significant distinction in the structure of sub-classes and/or when the semantics of their identification is not strong. The second solution is to consider only schemas corresponding to the sub-classes and duplicate the attributes of the super-class in these schemas; in particular it works if the super-class is defined as abstract. The third solution is to consider all the schemas corresponding to all classes of the hierarchy, resulting in a mesh of connected schemas and maintained at the expense of referential integrity rules. This solution has the advantage of avoiding duplication of information among different schemas, but suggests a dispersion of information by various schemas, and might involve a performance penalty in query operations or updating of data by requiring the execution of various join operations (i.e. "JOIN") and/or validation of referential integrity.

Alberto Manuel Rodrigues da Silva e Carlos Alberto Escaleira Videira, UML, metodologias e ferramentas CASE, 2ª Edição, Volume 1, Centro Atlântico Editora, Maio 2005.

Mapping Generalizations



```
# Superclass approach

media(id, type CHK {book, cd, dvd} ...)


# ER approach

media(id, ...)
book(id->media, ...)
cd(id->media, ...)
dvd(id->media, ...)


# Object Oriented

book(id, [media attributes], ...)
cd(id, [media attributes], ...)
dvd(id, [media attributes], ...)
```

A6. Indexes, Triggers, Transactions and Database Population

- ➔ This artifact contains the physical schema of the database,
  - ➔ the identification and characterization of the indexes,
  - ➔ the support of data integrity rules with triggers,
  - ➔ the definition of the database user-defined functions,
  - ➔ and the identification and characterization of the database transactions.
- ➔ This artifact also includes the complete database creation script, including all SQL code necessary to define all integrity constraints, indexes, triggers and transactions.
- ➔ Also, the database creation script and the database population script should be included as separate elements.

Indexes

A6. Indexes

- ➔ The **workload** is a study of the predicted system load, including an estimate on the number and growth of tuples in each relation.
- ➔ **Performance indexes** are applied to improve the performance of select queries.
  - ➔ At most, three performance indexes can be proposed, identifying the ones that are likely to have the biggest impact on the performance of the application.
- ➔ For each proposed index, it is necessary to indicate and justify the type chosen (B-tree, Hash, GiST, GIN), and also if clustering is recommended. As a last resource, controlled redundancy may be introduced (de-normalisation).
- ➔ The system being developed must provide full-text search features supported by PostgreSQL. Thus, it is necessary to **specify the fields where full-text search will be available** and the associated setup, namely all necessary configurations, indexes definitions and other relevant details.

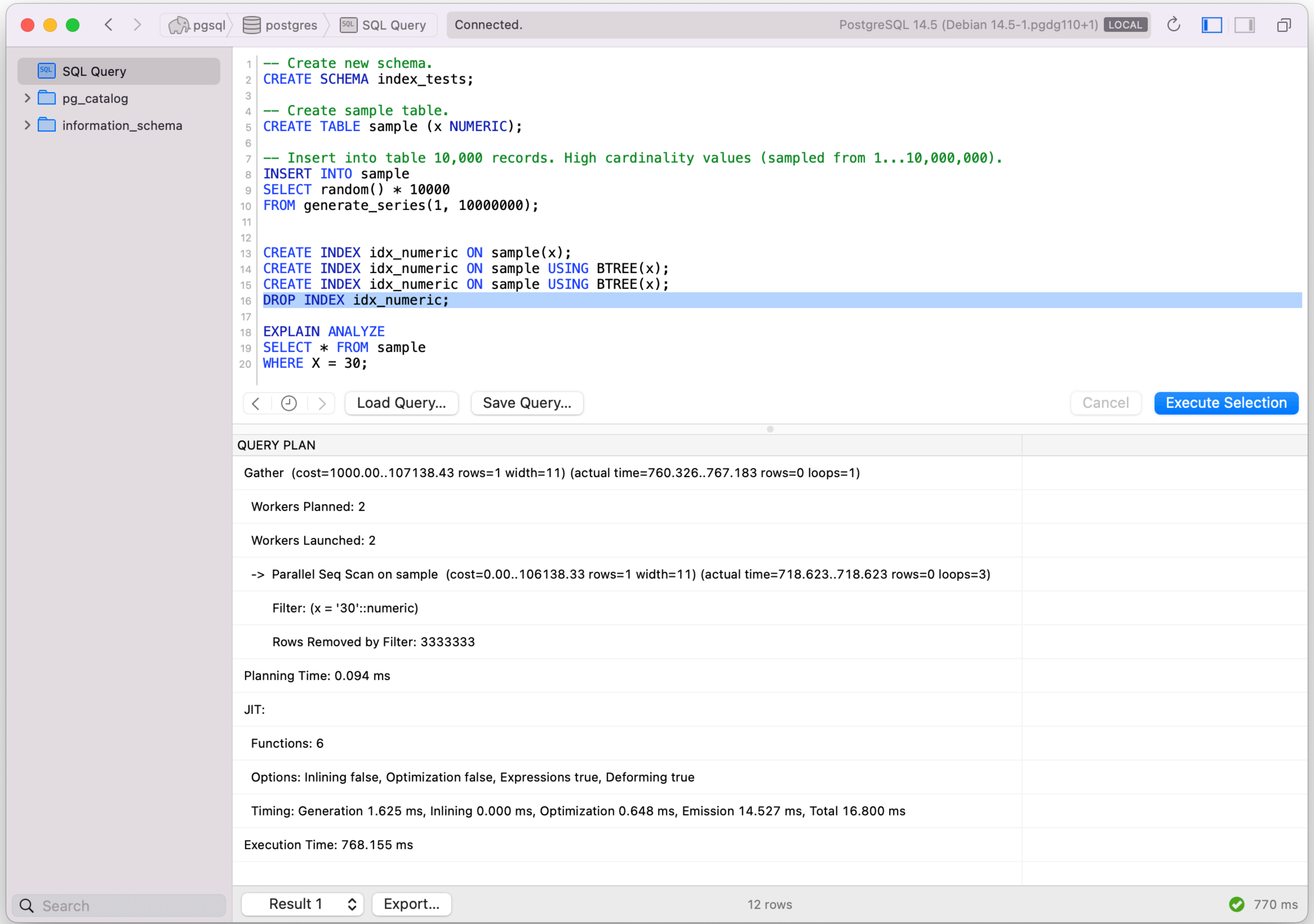
Indexes in PostgreSQL (1)

- ➔ Indexes are secondary data structures used to improve data access (*useful metaphor - the alphabetical back-of-the-book index*).
- ➔ Finding and retrieving specific rows is much faster with indexes, but they add an overhead to the execution.
  - ➔ Without indexes, tables are usually sequentially scanned to find the matching entry.
  - ➔ With indexes, the number of steps to find the matching records can be drastically reduced.
- ➔ Two main types:
  - ➔ B-tree indexes: use a tree-like data structure that maintains data sorted and allow for search, order, range search in log time.
  - ➔ Hash indexes: use a hash-function to map keys to values; are only considered when an equality operator is used (no sorting or ranges).

PostgreSQL example

```
-- Create new schema.
CREATE SCHEMA index_tests;

-- Create sample table.
CREATE TABLE sample (x NUMERIC);

-- Insert into table 10,000 records. High cardinality values (sampled from 1...10,000,000).
INSERT INTO sample
SELECT random() * 10000
FROM generate_series(1, 10000000);


CREATE INDEX idx_numeric ON sample(x);
CREATE INDEX idx_numeric ON sample USING BTREE(x);
CREATE INDEX idx_numeric ON sample USING HASH(x);
DROP INDEX idx_numeric;

EXPLAIN ANALYZE
SELECT * FROM sample
WHERE X = 30;
```

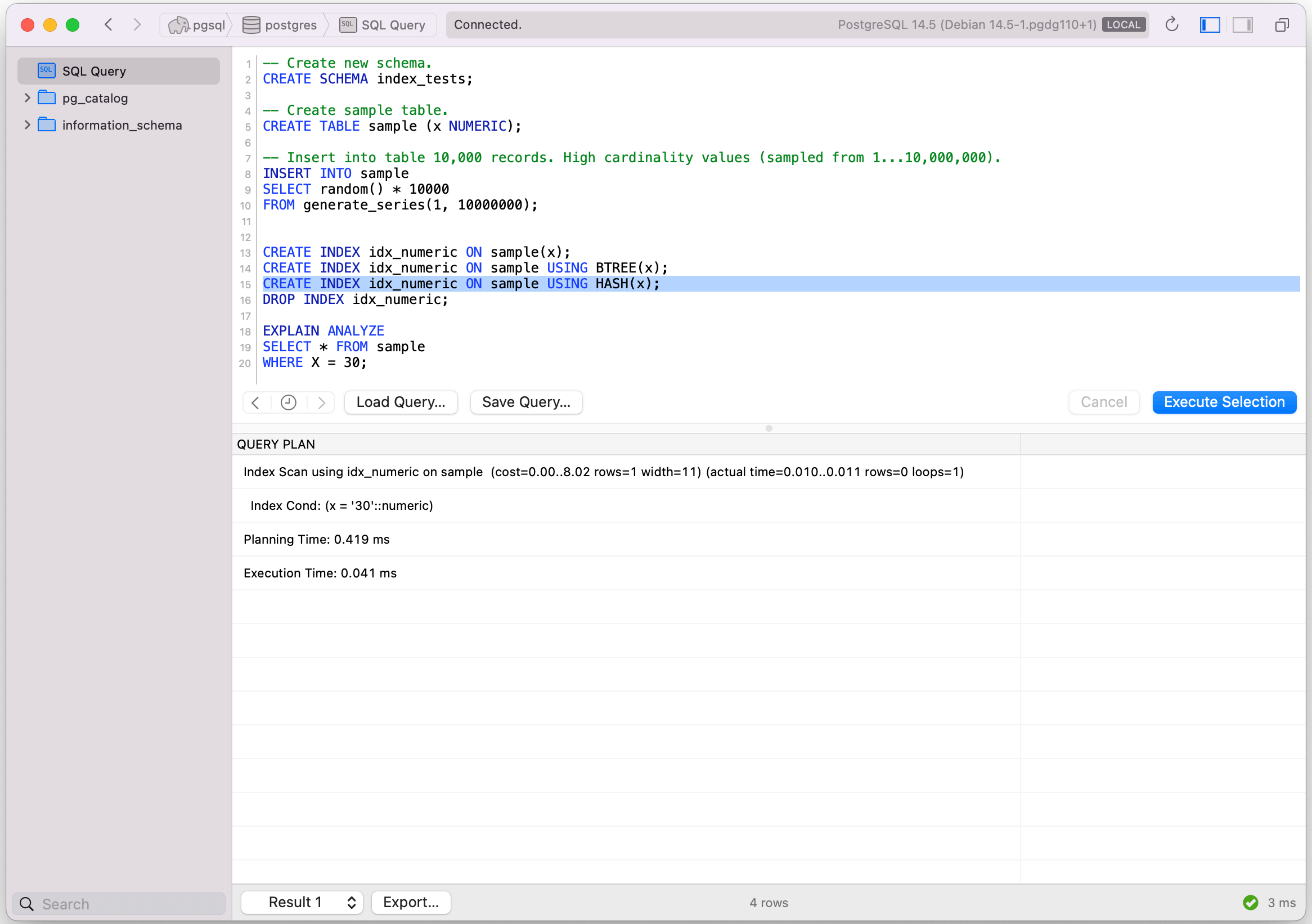
PostgreSQL example (no index)



```sql
-- Create new schema.
CREATE SCHEMA index_tests;

-- Create sample table.
CREATE TABLE sample (x NUMERIC);

-- Insert into table 10,000 records. High cardinality values (sampled from 1...10,000,000).
INSERT INTO sample
SELECT random() * 10000
FROM generate_series(1, 10000000);


CREATE INDEX idx_numeric ON sample(x);
CREATE INDEX idx_numeric ON sample USING BTREE(x);
CREATE INDEX idx_numeric ON sample USING BTREE(x);
DROP INDEX idx_numeric;

EXPLAIN ANALYZE
SELECT * FROM sample
WHERE X = 30;
```

```
QUERY PLAN
Gather  (cost=1000.00..107138.43 rows=1 width=11) (actual time=760.326..767.183 rows=0 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on sample  (cost=0.00..106138.33 rows=1 width=11) (actual time=718.623..718.623 rows=0 loops=3)
        Filter: (x = '30'::numeric)
        Rows Removed by Filter: 3333333
Planning Time: 0.094 ms
JIT:
  Functions: 6
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 1.625 ms, Inlining 0.000 ms, Optimization 0.648 ms, Emission 14.527 ms, Total 16.800 ms
Execution Time: 768.155 ms
```

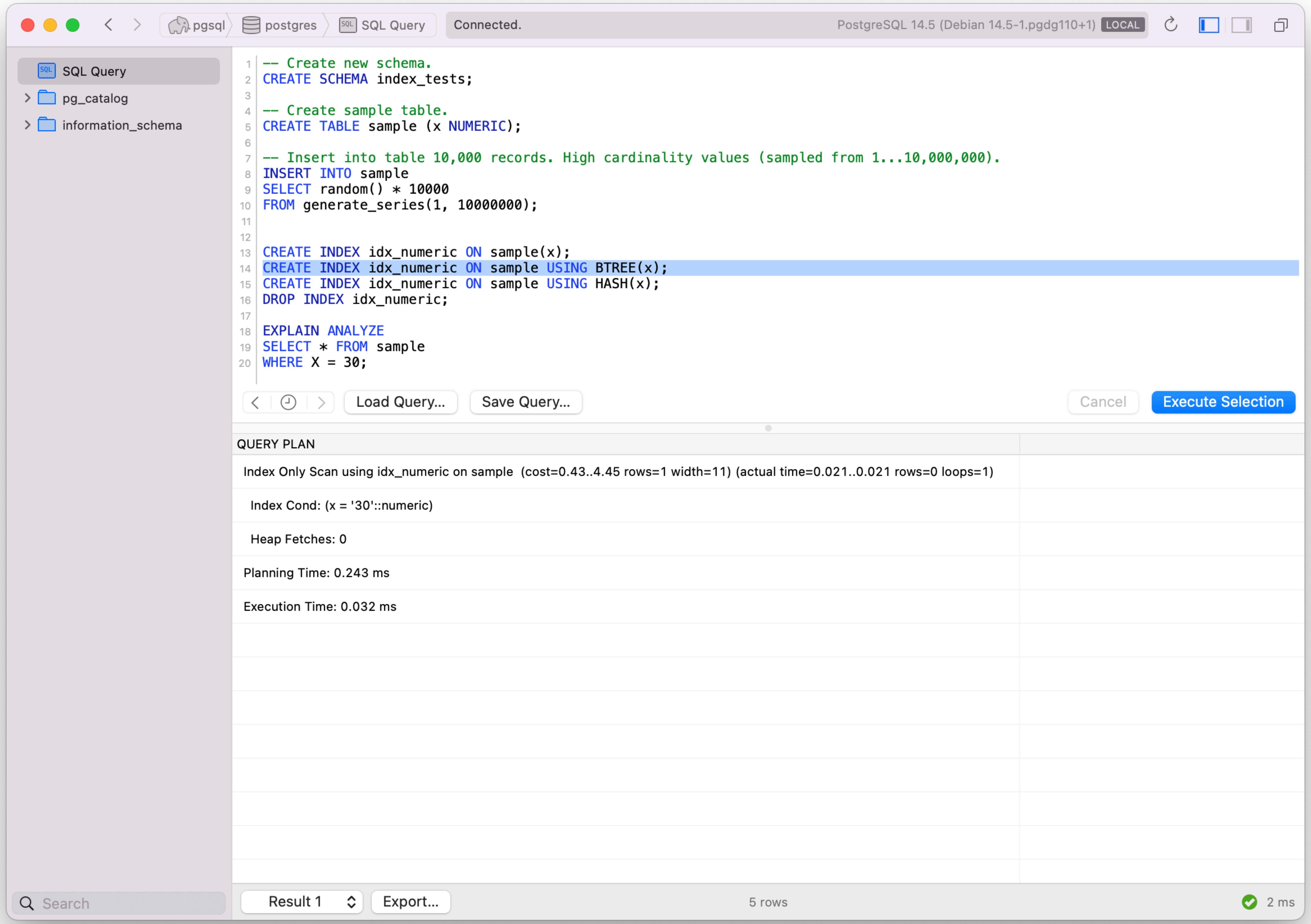
PostgreSQL example (hash)



```sql
-- Create new schema.
CREATE SCHEMA index_tests;

-- Create sample table.
CREATE TABLE sample (x NUMERIC);

-- Insert into table 10,000 records. High cardinality values (sampled from 1...10,000,000).
INSERT INTO sample
SELECT random() * 10000
FROM generate_series(1, 10000000);


CREATE INDEX idx_numeric ON sample(x);
CREATE INDEX idx_numeric ON sample USING BTREE(x);
CREATE INDEX idx_numeric ON sample USING HASH(x);
DROP INDEX idx_numeric;

EXPLAIN ANALYZE
SELECT * FROM sample
WHERE X = 30;
```

```
QUERY PLAN
Index Scan using idx_numeric on sample  (cost=0.00..8.02 rows=1 width=11) (actual time=0.010..0.011 rows=0 loops=1)
  Index Cond: (x = '30'::numeric)
Planning Time: 0.419 ms
Execution Time: 0.041 ms
```

PostgreSQL example (btree)



```sql
-- Create new schema.
CREATE SCHEMA index_tests;

-- Create sample table.
CREATE TABLE sample (x NUMERIC);

-- Insert into table 10,000 records. High cardinality values (sampled from 1...10,000,000).
INSERT INTO sample
SELECT random() * 10000
FROM generate_series(1, 10000000);


CREATE INDEX idx_numeric ON sample(x);
CREATE INDEX idx_numeric ON sample USING BTREE(x);
CREATE INDEX idx_numeric ON sample USING HASH(x);
DROP INDEX idx_numeric;

EXPLAIN ANALYZE
SELECT * FROM sample
WHERE X = 30;
```

```
QUERY PLAN
Index Only Scan using idx_numeric on sample  (cost=0.43..4.45 rows=1 width=11) (actual time=0.021..0.021 rows=0 loops=1)
  Index Cond: (x = '30'::numeric)
  Heap Fetches: 0
Planning Time: 0.243 ms
Execution Time: 0.032 ms
```

Indexes in PostgreSQL (2)

- ➔ Indexes can be created for more than one attribute (multicolumn).
  - ➔ CREATE INDEX name ON table (a, b);
  - ➔ Work when searching for both attributes simultaneously or just a. Not just b.
- ➔ Indexes can also be created for expressions.
  - ➔ SELECT * FROM test1 WHERE lower(col1) = 'value';
  - ➔ CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
- ➔ Index usage can be analyzed with the EXPLAIN command.

Unique Indexes

- ➔ Indexes can also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.
- ➔ CREATE UNIQUE INDEX name ON table (column [, ...]);
- ➔ When an index is declared unique, multiple table rows with equal indexed values are not allowed. Null values are not considered equal.
- ➔ PostgreSQL automatically creates a unique index when
  - ➔ a unique constraint is used or
  - ➔ a primary key is defined for a table.

Clustering

- ➔ Clustering a table, results in the physical re-ordering of data in disk based on the index information. To cluster a table, an index must already be defined.
- ➔ Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered.
- ➔ If needed, clustering can be set to run periodically using cron.
- ➔ Clustering will help when multiple records are read together and an index can group them. It will be irrelevant when single rows are randomly accessed in a table.

Cardinality

- ➔ The uniqueness of data values contained in a particular column.
- ➔ The lower the cardinality, the more duplicate values in the column.
- ➔ Examples:
  - ➔ high cardinality - primary key
  - ➔ medium cardinality - last name in a customer table
  - ➔ low cardinality - boolean column
- ➔ Cardinality is used by the PostgreSQL planner, amongst other statistics, to estimate the number of rows returned by a WHERE clause. This is then used to decide if, and what, indexes should be used.

Full Text Search

- ➔ How can you search for a work in text fields? And multiple words?
- ➔ Using the LIKE operator is not feasible
  - ➔ There is no linguistic support (e.g. singular / plural).
  - ➔ No ranking is provided, only a set of results.
  - ➔ Multiple words search is not supported.
  - ➔ There is no index support.
- ➔ It is necessary to index each word individually.
- ➔ This is called *full text search*, or simply *text search*, on PostgreSQL.
- ➔ Key first step — define what is a document in your search system and what information is relevant.

tsvector Type

- ➔ Text is broken into lexemes, a normalized representation of words, e.g. normalization includes converting to lowercase, identifying the stem, etc.
- ➔ The tsvector data type is used to store distinct lexemes.
  - ➔ SELECT to_tsvector('english', 'The quick brown fox jumps over the lazy dog')
  - ➔ 'brown':3 'dog':9 'fox':4 'jump':5 'lazi':8 'quick':2
- ➔ The function to_tsvectors returns a **tsvector** with duplicates removed, stop words removed, and the number of position of each lexeme recorded.

tsqueries Type

- ➔ Queries, i.e. searches, are represented as **tsqueries**.
- ➔ The to_tsquery and plainto_tsquery functions convert a text query to a tsquery, a structure optimized for searching tsvectors.
- ➔ SELECT plainto_tsquery('english','sail boats');
  - ➔ 'sail' & 'boat'
- ➔ SELECT plainto_tsquery('portuguese','o velho barco');
  - ➔ 'velh' & 'barc'

Matching tsqueries to tsvectors

- The @@ operator is used to **assert if a tsvector matches a tsquery**:
  - SELECT to_tsvector('portuguese','o velho barco') @@ plainto_tsquery('portuguese','barca');
    - t
  - SELECT to_tsvector('portuguese','o velho barco') @@ plainto_tsquery('portuguese','carro');
    - f
- SELECT title FROM posts
  WHERE to_tsvector('english', title || ' ' || body) @@ plainto_tsquery('english', 'jumping dog');

FTS Weights

- ➔ Sometimes we want to give more importance to some specific fields.
- ➔ We can use the **setweight** function to attach a weight to a certain tsvector.
- ➔ Weights go from 'A' (more important) to 'D' (less important).
- ➔ SELECT
  setweight(to_tsvector('english', 'The quick brown fox jumps over the lazy dog'), 'A') ||
  setweight(to_tsvector('english', 'An English language pangram. A sentence that contains all of the letters of the alphabet.'), 'B')
- ➔ 'alphabet':24B 'brown':3A 'contain':17B 'dog':9A 'english':11B 'fox':4A 'jump':5A 'languag':12B 'lazi':8A 'letter':21B 'pangram':13B' quick':2A 'sentenc':15B

Ranking FTS Results

- ➔ PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information:
  - ➔ how often the query terms appear in the document;
  - ➔ how close together the terms are in the document;
  - ➔ how important is the part of the document where they occur.
- ➔ Different applications might require additional information for ranking, e.g., document modification time. The built-in ranking functions are only examples.

Ranking FTS Results

- ➔ The ts_rank and ts_rank_cd functions, return a score for each returned row for a certain match between a tsquery and tsvector.
- ➔
```
SELECT  ts_rank(
setweight(to_tsvector('english', 'The quick brown fox jumps over the lazy dog'), 'A') ||
setweight(to_tsvector('english', 'An English language pangram. A sentence that contains all of the letters of the alphabet.'), 'B'),
plainto_tsquery('english', 'jumping dog')
)
```
- ➔
```
0.9524299
```
- ➔ You can also change the weights of the tsvector classes (A to D) and set how normalization, due to different document lengths, should be performed.

Pre-calculate FTS

- ➔ For performance reasons, we should consider adding a column to tables where FTS is to be performed containing the tsvector values of each row.
- ➔ This column should be updated whenever a row changes or is inserted. This can be done easily using a trigger.

```
CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    NEW.search = to_tsvector('english', NEW.title);
  END IF;
  IF TG_OP = 'UPDATE' THEN
      IF NEW.title <> OLD.title THEN
        NEW.search = to_tsvector('english', NEW.title);
      END IF;
  END IF;
  RETURN NEW;
END
$$ LANGUAGE 'plpgsql';
```

Indexing FTS

- To select all posts containing both ‘jumping’ and ‘dog’ we can use the following query

  -
    ```
    SELECT title FROM posts
    WHERE search @@ plainto_tsquery('english', 'jumping dog')
    ORDER BY ts_rank(search, plainto_tsquery('english', 'jumping dog')) DESC
    ```
- Note that ‘search’ is a pre-calculated column containing the tsvector of the columns we want to search.
- To improve the performance of our full text searches, we can use GIN or GiST indexes:
  - `CREATE INDEX search_idx ON posts USING GIN (search);`
  - `CREATE INDEX search_idx ON posts USING GIST (search);`
- “*As a rule of thumb, GIN indexes are best for static data because lookups are faster. For dynamic data, GiST indexes are faster to update.*”, PostgreSQL documentation.

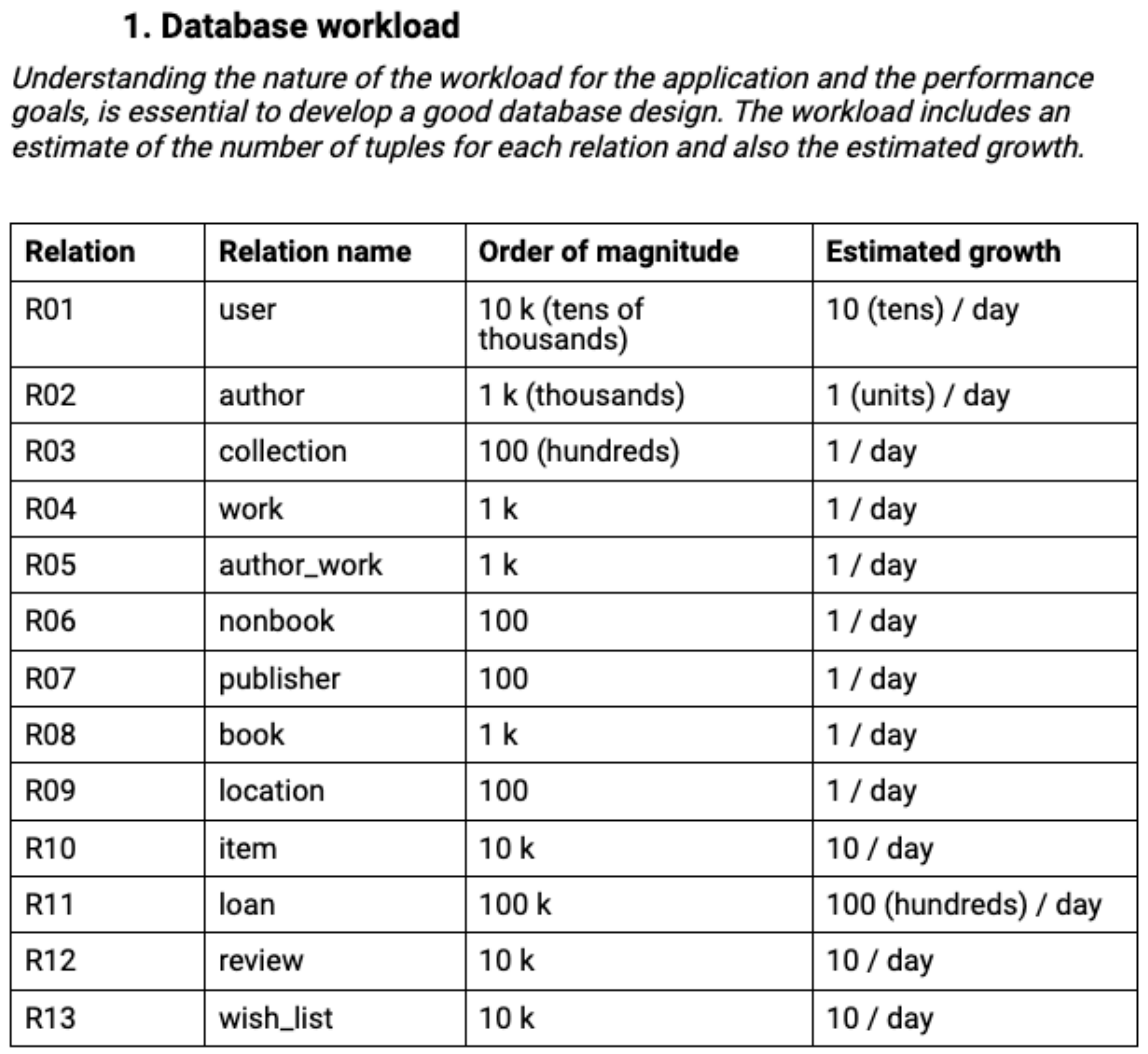
PostgreSQL example (full text search)

```
– Movies loaded from $ curl https://datasets.imdbws.com/title.akas.tsv.gz -o movies.tsv.gz


-- Create a new schema.
CREATE SCHEMA movies;

-- Create a new table with a single column 'title'.
CREATE TABLE movies (title TEXT);

-- Import data from the movies file. Consider only the 3rd column, and ignore the first line (i.e.
headers). ~>30 Millions
\copy movies FROM PROGRAM 'zcat movies.tsv.gz | cut -f3 | tail -n +2' WITH (FORMAT TEXT);

-- Add a new column for the tsvectors.
ALTER TABLE movies ADD COLUMN tsvectors TSVECTOR;

-- Create and store ts_vectors on the column named 'tsvectors'.
UPDATE movies SET tsvectors = to_tsvector('english', title);

-- …takes some time.

-- Create an index on the ts_vectors.
CREATE INDEX idx_titles ON movies USING GIN(tsvectors);


-- Query fast!
SELECT title, ts_rank(tsvectors, query) AS rank
FROM movies, plainto_tsquery('english','documentary nature') query
WHERE tsvectors @@ query
ORDER BY rank DESC;
```

PostgreSQL example (trigrams)

```sql
-- Load Trigram extension for fast trigram search.
CREATE EXTENSION pg_trgm;


CREATE INDEX idx_titles_trgm ON movies USING GIN(title gin_trgm_ops); -- takes some time.
DROP INDEX idx_titles_trgm;

-- EXPLAIN ANALYZE
SELECT title FROM movies WHERE title ILIKE 'spa%' LIMIT 10;
```

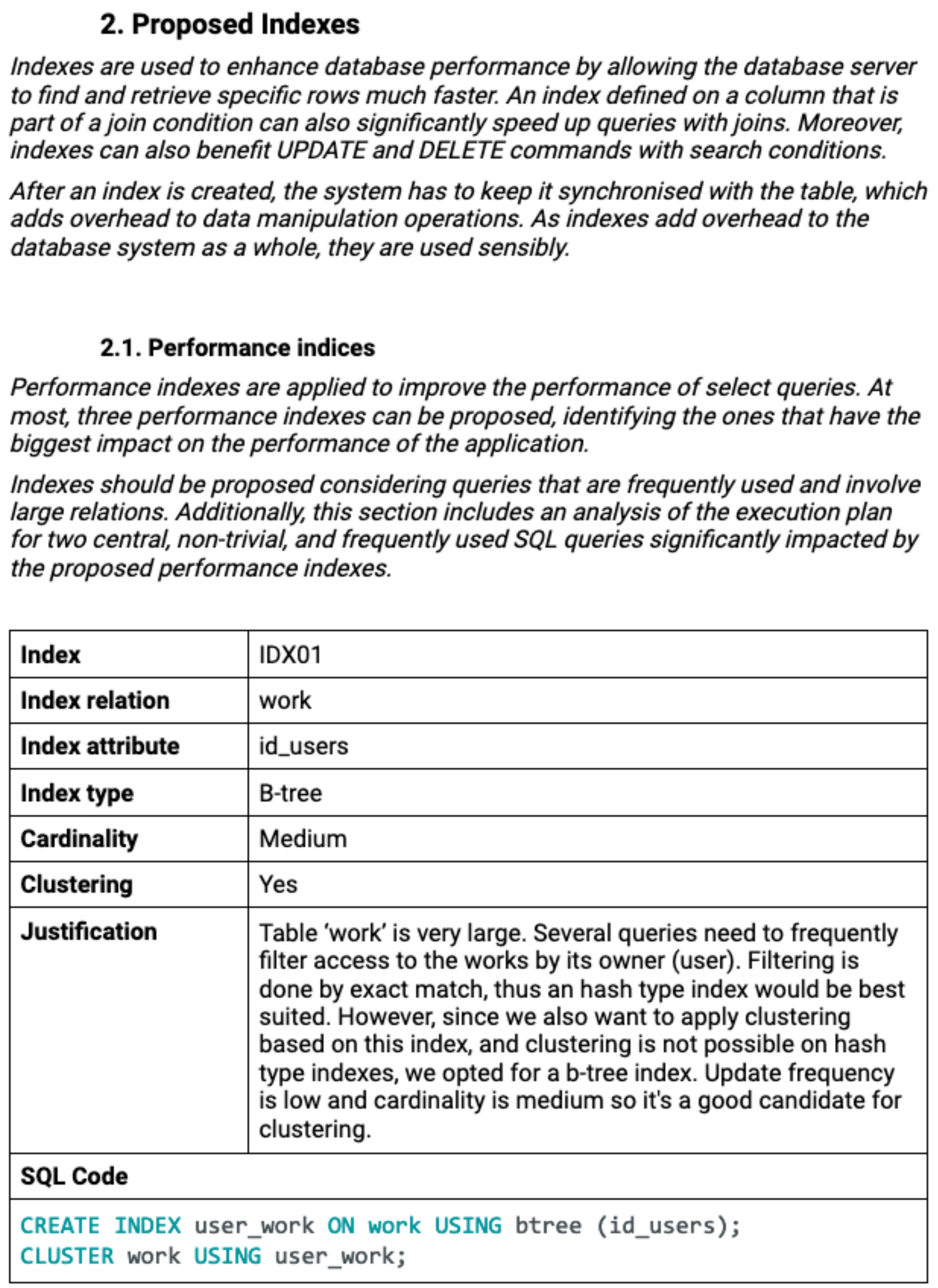
A6. MediaLibrary Indexes (Performance)



1. Database workload

Understanding the nature of the workload for the application and the performance goals, is essential to develop a good database design. The workload includes an estimate of the number of tuples for each relation and also the estimated growth.

Relation	Relation name	Order of magnitude	Estimated growth
R01	user	10 k (tens of thousands)	10 (tens) / day
R02	author	1 k (thousands)	1 (units) / day
R03	collection	100 (hundreds)	1 / day
R04	work	1 k	1 / day
R05	author_work	1 k	1 / day
R06	nonbook	100	1 / day
R07	publisher	100	1 / day
R08	book	1 k	1 / day
R09	location	100	1 / day
R10	item	10 k	10 / day
R11	loan	100 k	100 (hundreds) / day
R12	review	10 k	10 / day
R13	wish_list	10 k	10 / day



2. Proposed Indexes

Indexes are used to enhance database performance by allowing the database server to find and retrieve specific rows much faster. An index defined on a column that is part of a join condition can also significantly speed up queries with joins. Moreover, indexes can also benefit UPDATE and DELETE commands with search conditions.

After an index is created, the system has to keep it synchronised with the table, which adds overhead to data manipulation operations. As indexes add overhead to the database system as a whole, they are used sensibly.

2.1. Performance indices

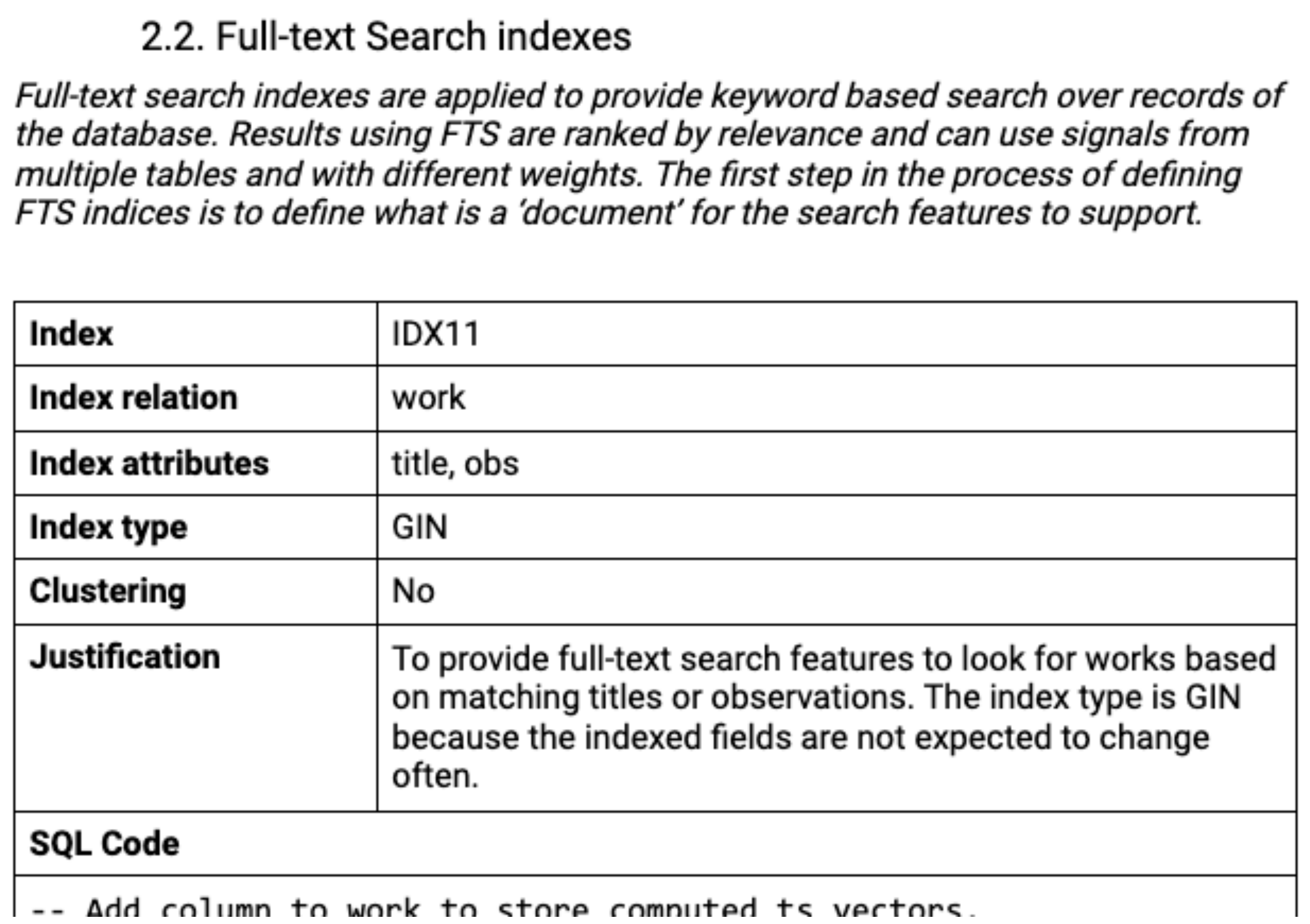
Performance indexes are applied to improve the performance of select queries. At most, three performance indexes can be proposed, identifying the ones that have the biggest impact on the performance of the application.

Indexes should be proposed considering queries that are frequently used and involve large relations. Additionally, this section includes an analysis of the execution plan for two central, non-trivial, and frequently used SQL queries significantly impacted by the proposed performance indexes.

Index	IDX01
Index relation	work
Index attribute	id_users
Index type	B-tree
Cardinality	Medium
Clustering	Yes
Justification	Table 'work' is very large. Several queries need to frequently filter access to the works by its owner (user). Filtering is done by exact match, thus an hash type index would be best suited. However, since we also want to apply clustering based on this index, and clustering is not possible on hash type indexes, we opted for a b-tree index. Update frequency is low and cardinality is medium so it's a good candidate for clustering.
SQL Code	

```
CREATE INDEX user_work ON work USING btree (id_users);
CLUSTER work USING user_work;
```

A6. MediaLibrary Indexes (Full Text Search)

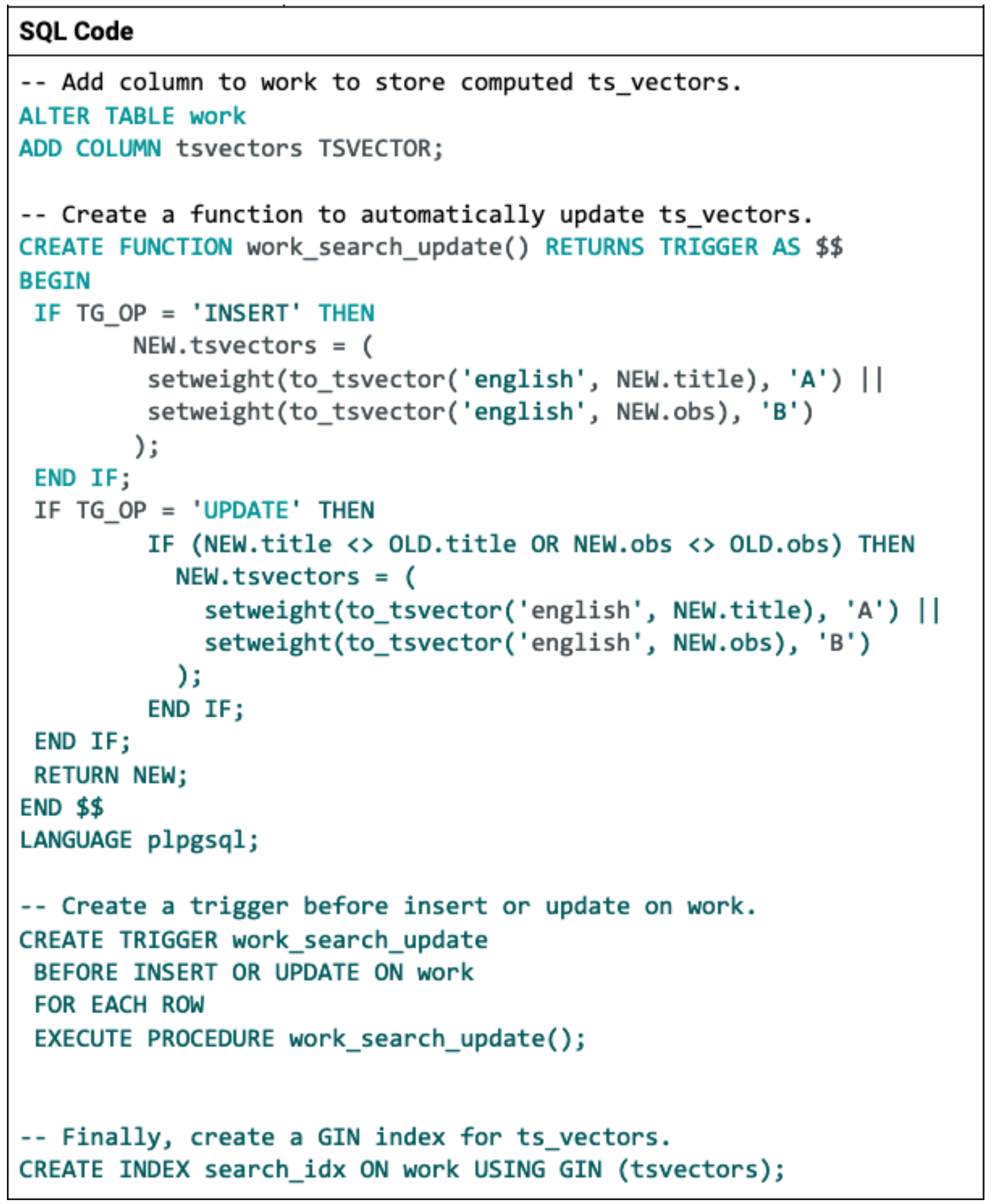


2.2. Full-text Search indexes

Full-text search indexes are applied to provide keyword based search over records of the database. Results using FTS are ranked by relevance and can use signals from multiple tables and with different weights. The first step in the process of defining FTS indices is to define what is a 'document' for the search features to support.

Index	IDX11
Index relation	work
Index attributes	title, obs
Index type	GIN
Clustering	No
Justification	To provide full-text search features to look for works based on matching titles or observations. The index type is GIN because the indexed fields are not expected to change often.
SQL Code	

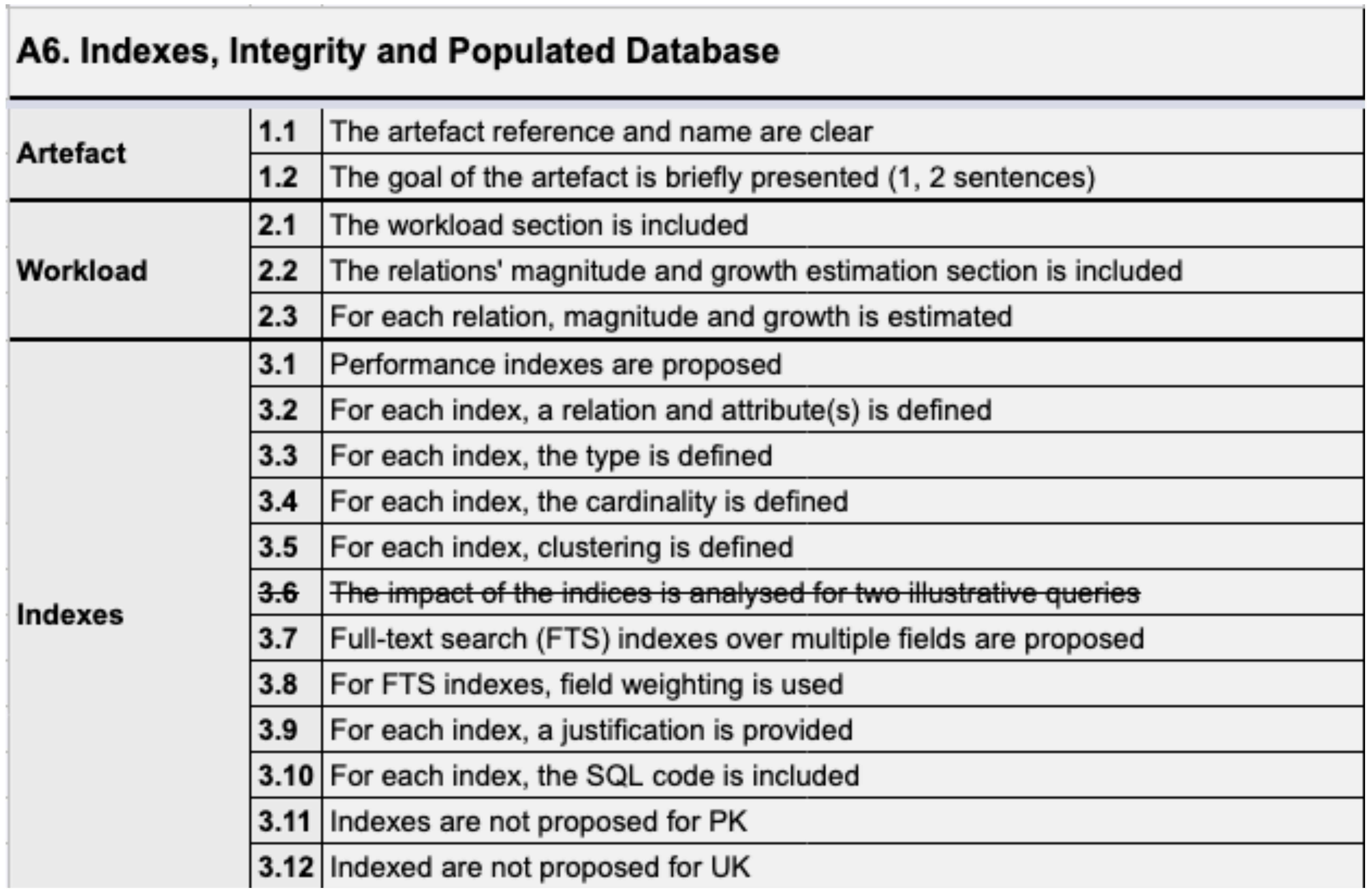
```
-- Add column to work to store computed ts_vectors.
```



SQL Code

```
-- Add column to work to store computed ts_vectors.
ALTER TABLE work
ADD COLUMN tsvectors TSVECTOR;

-- Create a function to automatically update ts_vectors.
CREATE FUNCTION work_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.title), 'A') ||
         setweight(to_tsvector('english', NEW.obs), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.title <> OLD.title OR NEW.obs <> OLD.obs) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.title), 'A') ||
             setweight(to_tsvector('english', NEW.obs), 'B')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

-- Create a trigger before insert or update on work.
CREATE TRIGGER work_search_update
 BEFORE INSERT OR UPDATE ON work
 FOR EACH ROW
 EXECUTE PROCEDURE work_search_update();


-- Finally, create a GIN index for ts_vectors.
CREATE INDEX search_idx ON work USING GIN (tsvectors);
```

A6. Indexes Checklist



A6. Indexes, Integrity and Populated Database		
Artefact	1.1	The artefact reference and name are clear
	1.2	The goal of the artefact is briefly presented (1, 2 sentences)
Workload	2.1	The workload section is included
	2.2	The relations' magnitude and growth estimation section is included
	2.3	For each relation, magnitude and growth is estimated
Indexes	3.1	Performance indexes are proposed
	3.2	For each index, a relation and attribute(s) is defined
	3.3	For each index, the type is defined
	3.4	For each index, the cardinality is defined
	3.5	For each index, clustering is defined
	~~3.6~~	~~The impact of the indices is analysed for two illustrative queries~~
	3.7	Full-text search (FTS) indexes over multiple fields are proposed
	3.8	For FTS indexes, field weighting is used
	3.9	For each index, a justification is provided
	3.10	For each index, the SQL code is included
	3.11	Indexes are not proposed for PK
	3.12	Indexed are not proposed for UK

Triggers and User Defined Functions

A6. Triggers and User Defined Functions

- ➔ To enforce integrity rules that cannot be achieved in a simpler way, the necessary triggers are identified and described by presenting the event, the condition, and the activation code.
- ➔ User-defined functions, and trigger procedures, that add control structures to the SQL language, or perform complex computations, are identified and described to be trusted by the database server.
- ➔ Every kind of function (SQL functions, Stored procedures, Trigger procedures) can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.
- ➔ Common examples:
  - ➔ User cannot post in groups when he is not a member;
  - ➔ When a vote is cast, the rating (karma) of the author is updated;
  - ➔ When *an event happens*, relevant notifications are sent.

User-Defined Functions

- ➔ A user-defined function provides a mechanism for extending the functionality of the database server by adding a function.

- ➔ Advantages of using *stored procedures*:
  - ➔ Reduce the number of round trips between application and database server
  - ➔ Increase the application performance, because user-defined functions are pre-compiled and stored in the database server uses its full-power
  - ➔ Be able to be reused in many applications
- ➔ Disadvantages of *stored procedures*:
  - ➔ Slow software development because it requires specialized skills that many developers do not possess (PL/SQL)
  - ➔ Make it difficult to manage versions and hard to debug
  - ➔ Less portable code to other database management systems (MySQL, SQL Server, PostgreSQL, Oracle, DB2)

UDF Example

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $name$
DECLARE
declaration;
[...]
BEGIN
< function_body >
[...]
RETURN { variable_name | VALUE }
END;
$name$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION totalRecords ()
RETURNS INTEGER AS $total$
DECLARE
  total INTEGER;
BEGIN
  SELECT COUNT(*) INTO total FROM company;
  RETURN total;
END;
$total$ LANGUAGE plpgsql;

SELECT totalRecords();
```

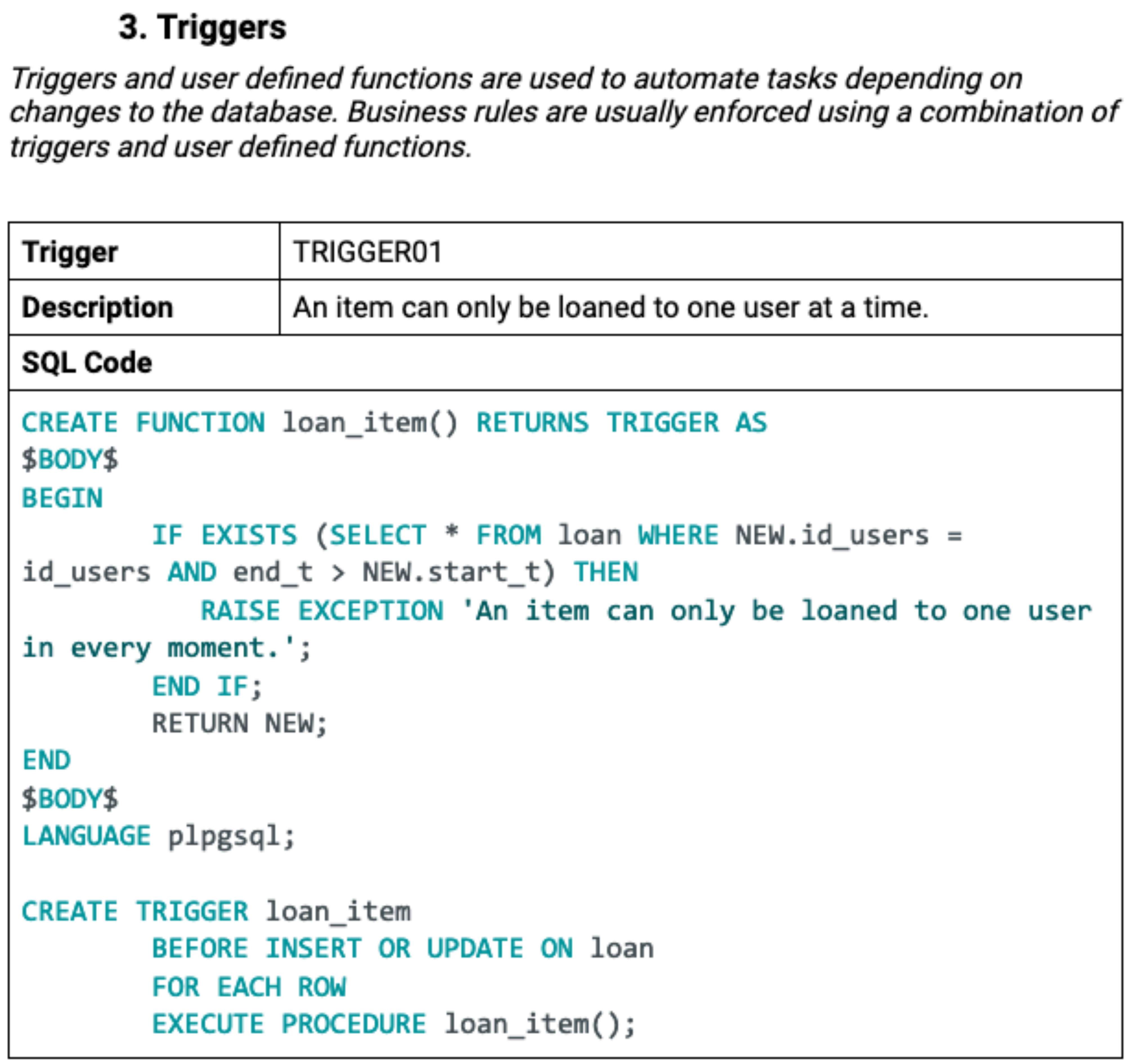
Triggers

- ➔ Triggers are *event-condition-action* rules:
  - ➔ Event, a change to the database that activates the trigger
  - ➔ Condition, a query or test that is run when the trigger is activated
  - ➔ Action, a procedure that is executed when the trigger is activated and its condition is true
- ➔ The action can be executed *before, after or instead of the trigger* event
- ➔ The action may refer the *new values and old values of records inserted*, updated or deleted in the trigger event
- ➔ The programmer specifies that the action is performed:
  - ➔ once for each modified record (FOR EACH ROW)
  - ➔ once for all records that are changed on a database operation

Triggers Example

```
CREATE FUNCTION loan_item() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM loan WHERE NEW.id_item = id_item AND end_t > NEW.start_t) THEN
           RAISE EXCEPTION 'An item can only be loaned to one user in every moment.';
        END IF;
        RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;


CREATE TRIGGER loan_item
        BEFORE INSERT OR UPDATE ON loan
        FOR EACH ROW
        EXECUTE PROCEDURE loan_item();
```

A6. MediaLibrary Triggers



3. Triggers

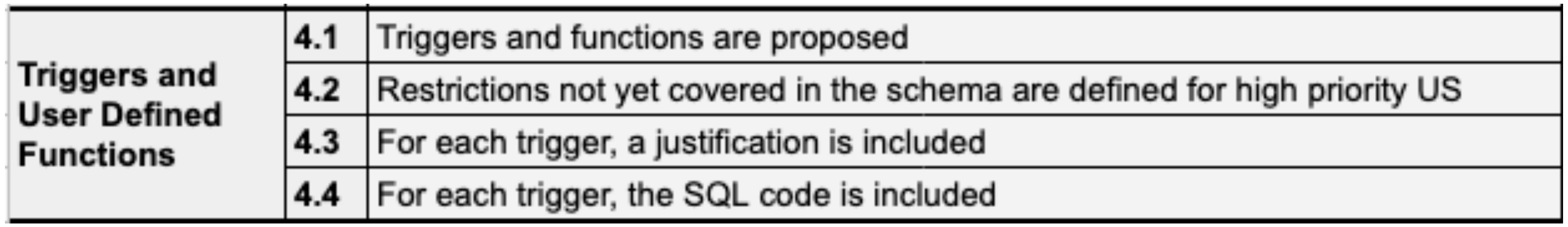
Triggers and user defined functions are used to automate tasks depending on changes to the database. Business rules are usually enforced using a combination of triggers and user defined functions.

Trigger	TRIGGER01
Description	An item can only be loaned to one user at a time.

SQL Code

```
CREATE FUNCTION loan_item() RETURNS TRIGGER AS
$BODY$
BEGIN
        IF EXISTS (SELECT * FROM loan WHERE NEW.id_users =
id_users AND end_t > NEW.start_t) THEN
           RAISE EXCEPTION 'An item can only be loaned to one user
in every moment.';
        END IF;
        RETURN NEW;
END
$BODY$
LANGUAGE plpgsql;

CREATE TRIGGER loan_item
        BEFORE INSERT OR UPDATE ON loan
        FOR EACH ROW
        EXECUTE PROCEDURE loan_item();
```

A6. Triggers Checklist



Triggers and User Defined Functions	**4.1**	Triggers and functions are proposed
	4.2	Restrictions not yet covered in the schema are defined for high priority US
	4.3	For each trigger, a justification is included
	4.4	For each trigger, the SQL code is included

Transactions

A6. Transactions

- ➔ Transactions bundle multiple steps into a single, all-or-noting operation, ensuring data integrity with concurrent accesses.
- ➔ For each necessary transaction, include:
  - ➔ Justification
  - ➔ Isolation level
  - ➔ SQL code to create it
- ➔ Common examples:
  - ➔ insert data (e.g. generalizations, latest generated PK)
  - ➔ delete data (e.g. user deletes account)
  - ➔ checkout purchase (move products from cart to purchase)
  - ➔ *many more*

Transactions Overview

Why Transactions

- ➔ Most databases operate in a multi-user context.
  - ➔ Many applications can access the data in parallel, both for reading and writing.
  - ➔ *E.g., a transfer between two bank accounts (i.e., two operations).*
- ➔ Even in single access contexts, complex operations need to be executed as a unit.
  - ➔ The system must be resilient in face of severe failures, such as power outages, disk or network failures.
  - ➔ *E.g., checking out a shopping cart requires moving items from the cart to the purchase.*
- ➔ Users should never be confronted with an inconsistent database state.
- ➔ **Transactions are a solution for both concurrency and failures.**

ACID Properties

- ➔ DBMS must support ACID properties.
  - ➔ **Atomicity** guarantees that multiple operations are treated as an indivisible unit.
  - ➔ **Consistency** guarantees that the database moves from one consistent state into another consistent state (clients must ensure the application logic follows the domain rules, being the primary responsible for the consistency of the system).
  - ➔ **Isolation** guarantees that the outcome of multiple transactions executed concurrently is the same as if every transaction was executed in isolation.
  - ➔ **Durability** refers to the fact that the effects of a committed transaction should be persisted into the database.

Transactions

- ➔ A transactions is a set of database operations that is considered as a single unit.
  - ➔ A transaction either succeeds or fails in its entirety.
  - ➔ The intermediate states between the steps of a transaction are not visible to other concurrent transactions.
  - ➔ Transactions renders a database from one consistent state into another consistent state.
- ➔ A DBMS supporting transactions includes:
  - ➔ Transaction management, i.e. features to support and control transaction execution.
  - ➔ Recovery, i.e. features to support recovery from failures.
  - ➔ Concurrency control, i.e. features to control concurrent execution of transactions.

Sample Transaction in PostgreSQL Syntax

```sql
-- Explicitly start the transaction.
BEGIN;

UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';

-- Savepoints can be used to return to during the transaction.
SAVEPOINT my_savepoint;

UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';

-- Rollback to the savepoint.
-- The previous UPDATE statement is discarded.
ROLLBACK TO my_savepoint;

UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';

-- End the transaction.
COMMIT;
```

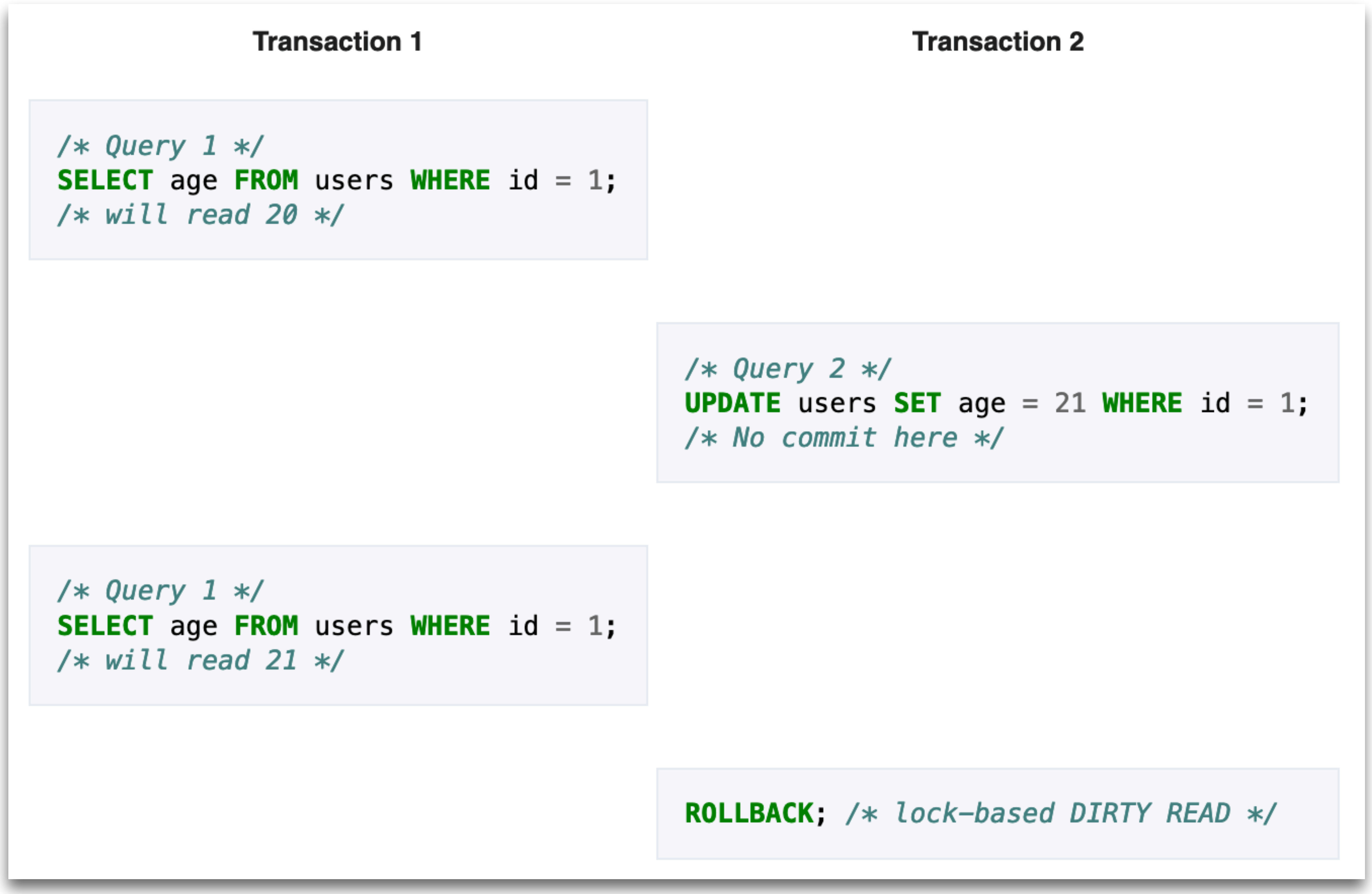
Concurrency Problems

- ➔ Different problems can occur when concurrent transactions execute.
- ➔ **Dirty reads**, a transaction reads data written by a concurrent uncommitted transaction.
- ➔ **Non-repeatable reads**, a transaction re-reads data and finds that data has been modified by another transaction. The same query returns different results during a transaction.
- ➔ **Phantom reads**, a transaction re-executes a query and finds that the results have changed by another transaction. The values have not changed but different rows are being returned.
- ➔ **Serialization anomaly**, the result of committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

Dirty reads

➔ Dirty reads, a transaction reads data written by a concurrent uncommitted transaction.

➔ Transaction 1 will read uncommitted data that was latter rolled back by Transaction 2.



Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 20 */
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
/* No commit here */
```

Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 21 */
```

Transaction 2

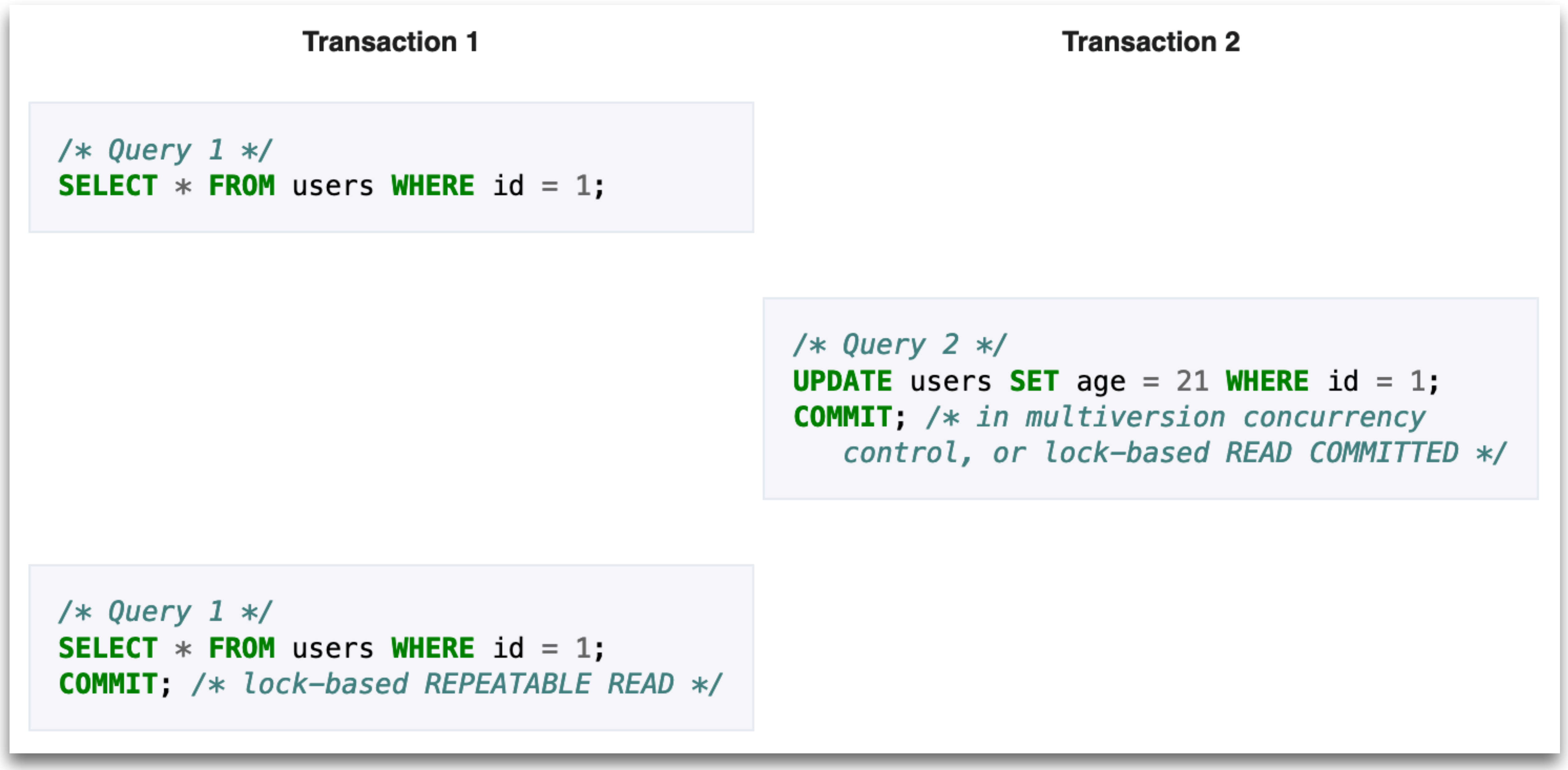
```
ROLLBACK; /* lock-based DIRTY READ */
```

Example from Wikipedia.
https://en.wikipedia.org/wiki/Isolation_(database_systems)

Non-repeatable reads

- ➔ Non-repeatable reads, a transaction re-reads data and finds that data has been modified by another transaction. The same query returns different results during a transaction.

- ➔ Transaction 1 will read the same data twice and obtain different values, because a concurrent transaction committed changes during the execution of Transaction 1.



Transaction 1

```
/* Query 1 */
SELECT * FROM users WHERE id = 1;
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
COMMIT; /* in multiversion concurrency
   control, or lock-based READ COMMITTED */
```

Transaction 1

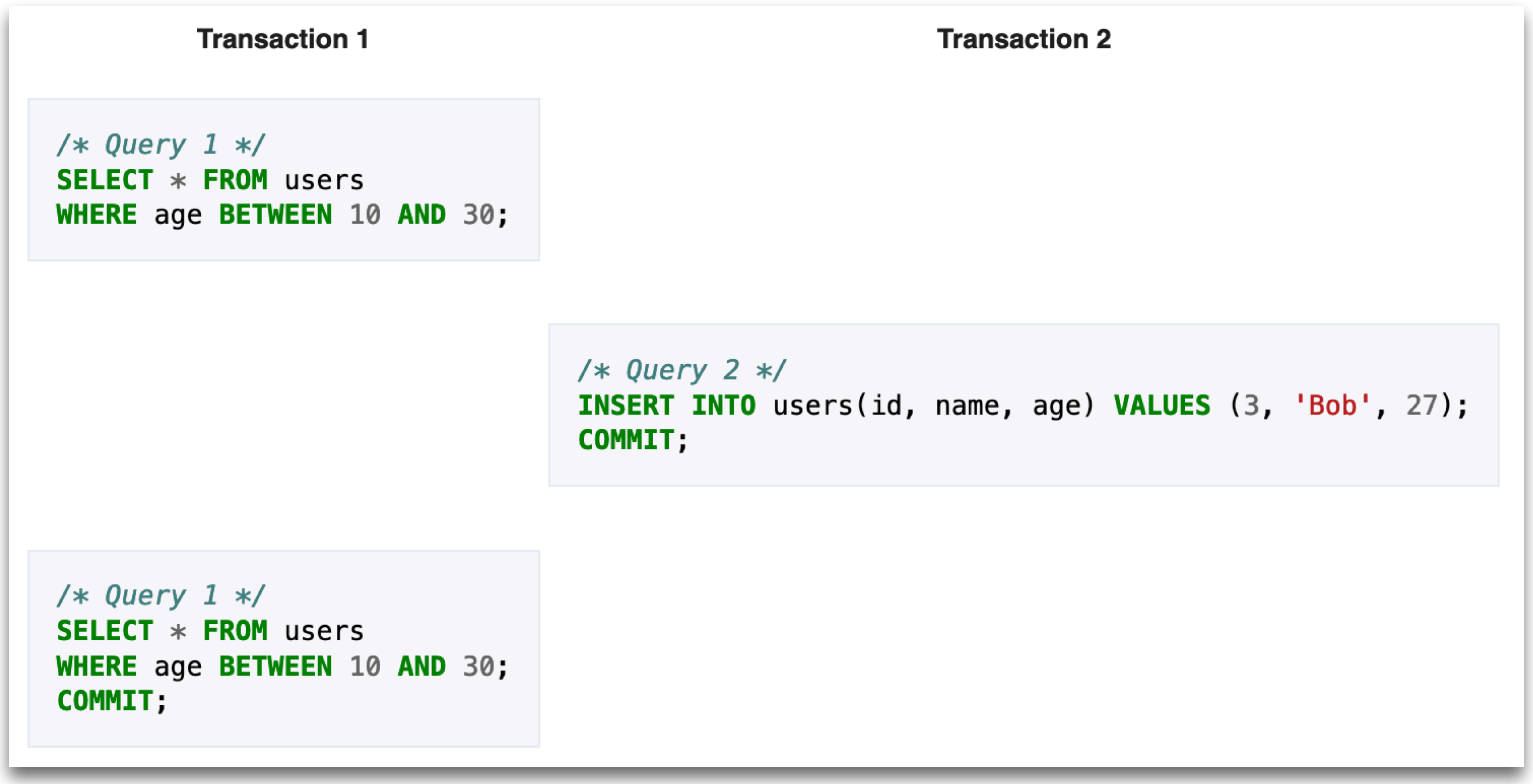
```
/* Query 1 */
SELECT * FROM users WHERE id = 1;
COMMIT; /* lock-based REPEATABLE READ */
```

Example from Wikipedia.
https://en.wikipedia.org/wiki/Isolation_(database_systems)

Phantom reads

➔ Phantom reads, a transaction re-executes a query and finds that the results have changed by another transaction. The values have not changed but different rows are being returned.

➔ Different values are read by Transaction 1 for the same query because Transaction 2 added new data (no changes made to existing data).



Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users(id, name, age) VALUES (3, 'Bob', 27);
COMMIT;
```

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
COMMIT;
```

Example from Wikipedia.
https://en.wikipedia.org/wiki/Isolation_(database_systems)

Serialization anomaly

➔ Serialization anomaly, the result of committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

➔ The result is inconsistent with all the possible orderings of the two transactions.

➔ Using isolation level serializable in PostgreSQL solves this problem. When committing, one of the transactions would fail.

Example from Wikipedia.
https://en.wikipedia.org/wiki/Isolation_(database_systems)

```
/* Transaction #1 */


/* Query 1 */
SELECT * FROM bank_teller;

 id | owner | balance
----+-------+---------
  1 | alice |    +100
  2 | bob   |    +120

/* Query 2 */
INSERT INTO bank_teller (owner, balance)
VALUES ('sum', 220);

/* Query 3 */
SELECT * FROM bank_teller;

 id | owner | balance
----+-------+---------
  1 | alice |    +100
  2 | bob   |    +120
  3 | sum   |    +220
```

```
/* Transaction #2 */


/* Query 1 */
SELECT * FROM bank_teller;

 id | owner | balance
----+-------+---------
  1 | alice |    +100
  2 | bob   |    +120

/* Query 2 */
INSERT INTO bank_teller (owner, balance)
VALUES ('sum', 220);

/* Query 3 */
SELECT * FROM bank_teller;

 id | owner | balance
----+-------+---------
  1 | alice |    +100
  2 | bob   |    +120
  3 | sum   |    +220
```

```
/* Result */

 id | owner | balance
----+-------+---------
  1 | alice |    +100
  2 | bob   |    +120
  3 | sum   |    +220
  4 | sum   |    +220
```

```
/* Expected */

 id | owner | balance
----+-------+---------
  1 | alice |    +100
  2 | bob   |    +120
  3 | sum   |    +220
  4 | sum   |    +440
```

Transaction Isolation

- ➔ DBMS offer different isolation levels to deal with concurrency problems (achieved mostly by locking access to tables).
- ➔ Stricter or looser isolation levels will allow less or more concurrent accesses.
- ➔ We should aim to the less restrictive isolation level that still guarantees that data is consistent
  - ➔ Advice: declare transactions as READ ONLY when possible.
- ➔ Isolation levels (from lowest/relaxed to highest/stricter):
  - ➔ **read uncommitted**, records still uncommitted can be read; Typically only allowed for read-only transactions.
    In PostgreSQL read uncommitted behaves as read committed.
  - ➔ **read committed** (*default in PostgreSQL*), transactions see only data that is committed at the moment it is read; it never sees data changed, and uncommitted, by other concurrent transactions. Uses long-term write locks and short-term read locks.
  - ➔ **repeatable read**, transactions only see data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. Uses long-term locks for both writes and reads.
  - ➔ **serializable**, transactions see only data committed before the transaction began and never sees uncommitted data or changes;

Transaction Isolation Levels in PostgreSQL (11)

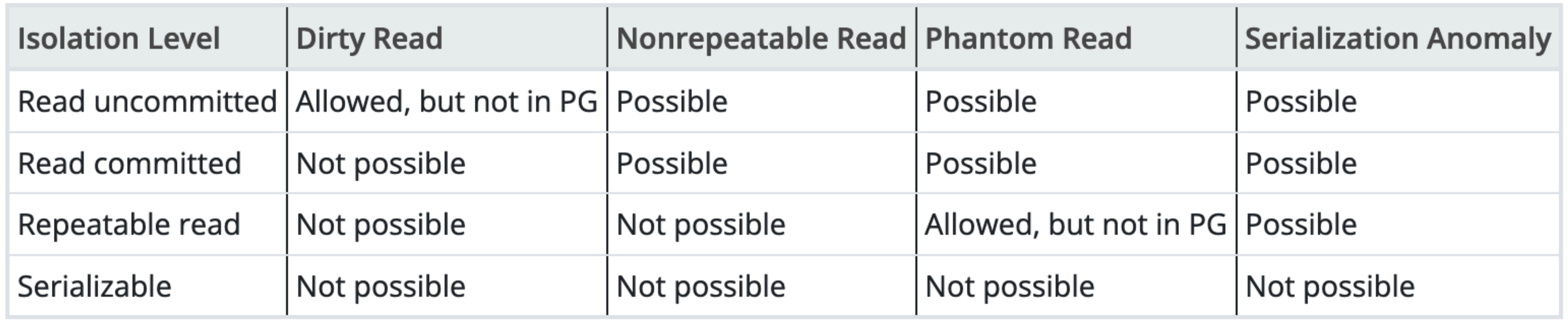


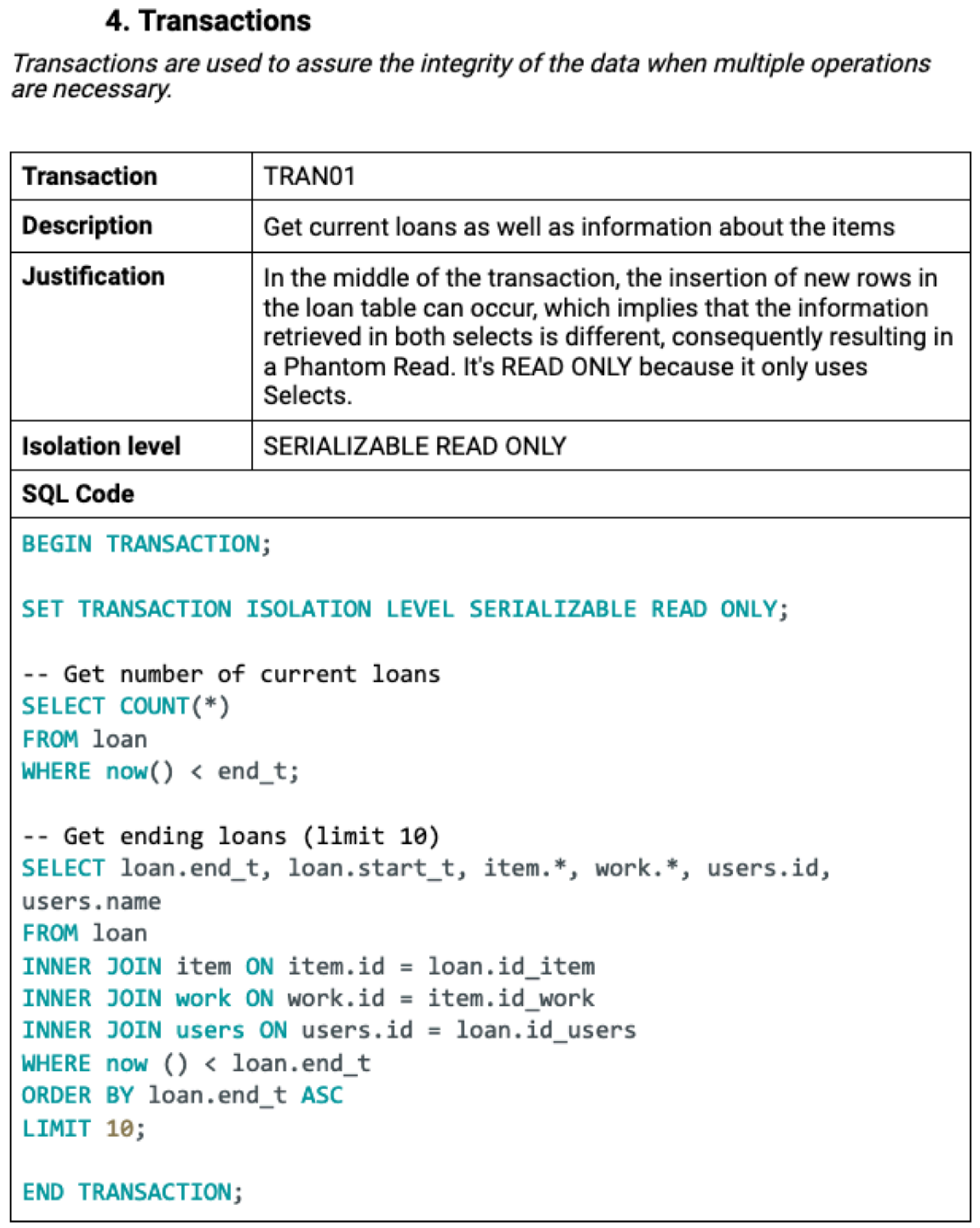
Table 13.1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Transactions Summary

- ➔ The SQL standard default isolation level is serializable.
- ➔ Weaker isolation levels:
  - ➔ Increased concurrency + decreased overhead = increased performance
  - ➔ Weaker consistency guarantees
  - ➔ PostgreSQL default isolation level level is read committed.
- ➔ Stronger isolation levels:
  - ➔ Reduced concurrency + increased overhead = decreased performance
  - ➔ Strong consistency guarantees
- ➔ NOSQL systems abandon the ACID properties to achieve large scale distributed performance.

A6. MediaLibrary Transactions



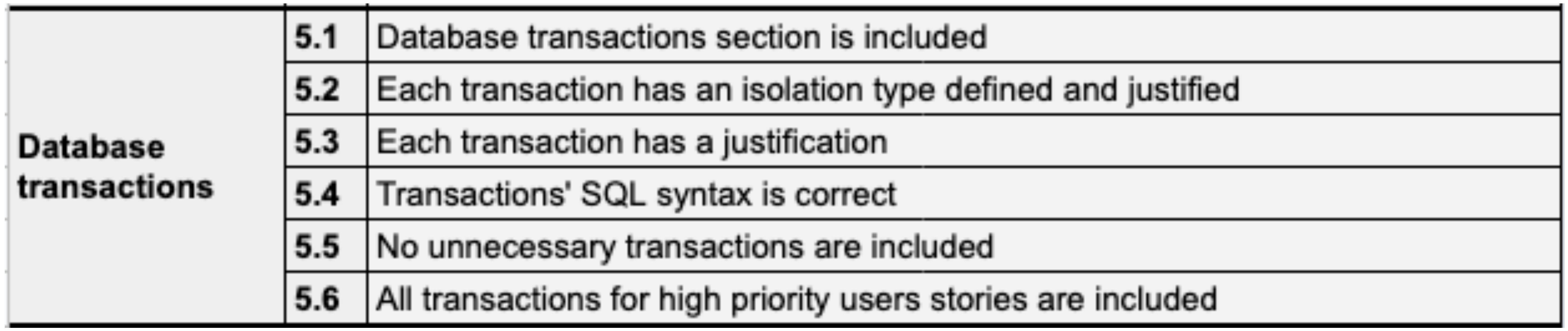
4. Transactions

Transactions are used to assure the integrity of the data when multiple operations are necessary.

Transaction	TRAN01
Description	Get current loans as well as information about the items
Justification	In the middle of the transaction, the insertion of new rows in the loan table can occur, which implies that the information retrieved in both selects is different, consequently resulting in a Phantom Read. It's READ ONLY because it only uses Selects.
Isolation level	SERIALIZABLE READ ONLY
SQL Code	

```sql
BEGIN TRANSACTION;

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY;

-- Get number of current loans
SELECT COUNT(*)
FROM loan
WHERE now() < end_t;

-- Get ending loans (limit 10)
SELECT loan.end_t, loan.start_t, item.*, work.*, users.id,
users.name
FROM loan
INNER JOIN item ON item.id = loan.id_item
INNER JOIN work ON work.id = item.id_work
INNER JOIN users ON users.id = loan.id_users
WHERE now () < loan.end_t
ORDER BY loan.end_t ASC
LIMIT 10;

END TRANSACTION;
```

A6. Transactions Checklist



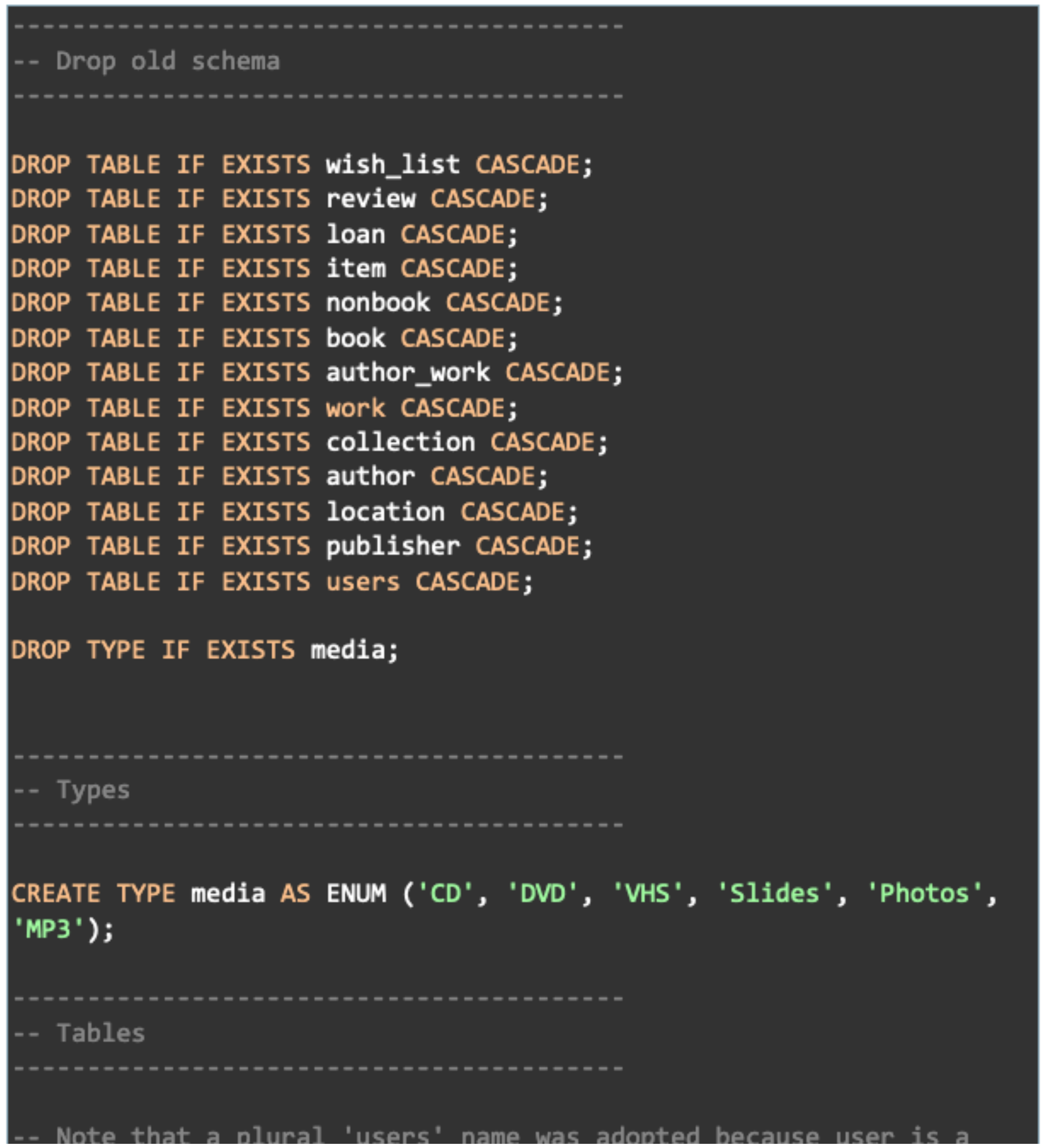
Database transactions	**5.1**	Database transactions section is included
	5.2	Each transaction has an isolation type defined and justified
	5.3	Each transaction has a justification
	5.4	Transactions' SQL syntax is correct
	5.5	No unnecessary transactions are included
	5.6	All transactions for high priority users stories are included

A6. Database Population

- ➔ The EBD Component includes two SQL scripts
  - ➔ **Database creation script**, including SQL creation statements for all tables, key constraints, performance indexes, full text search indexes, triggers, user defined functions;
  - ➔ **Database population script**, including SQL insert statements to populate a database with test data with an amount of tuples suitable for testing and with plausible values for each field type.
- ➔ These scripts must run, *as-is*, in the production PostgreSQL environment.

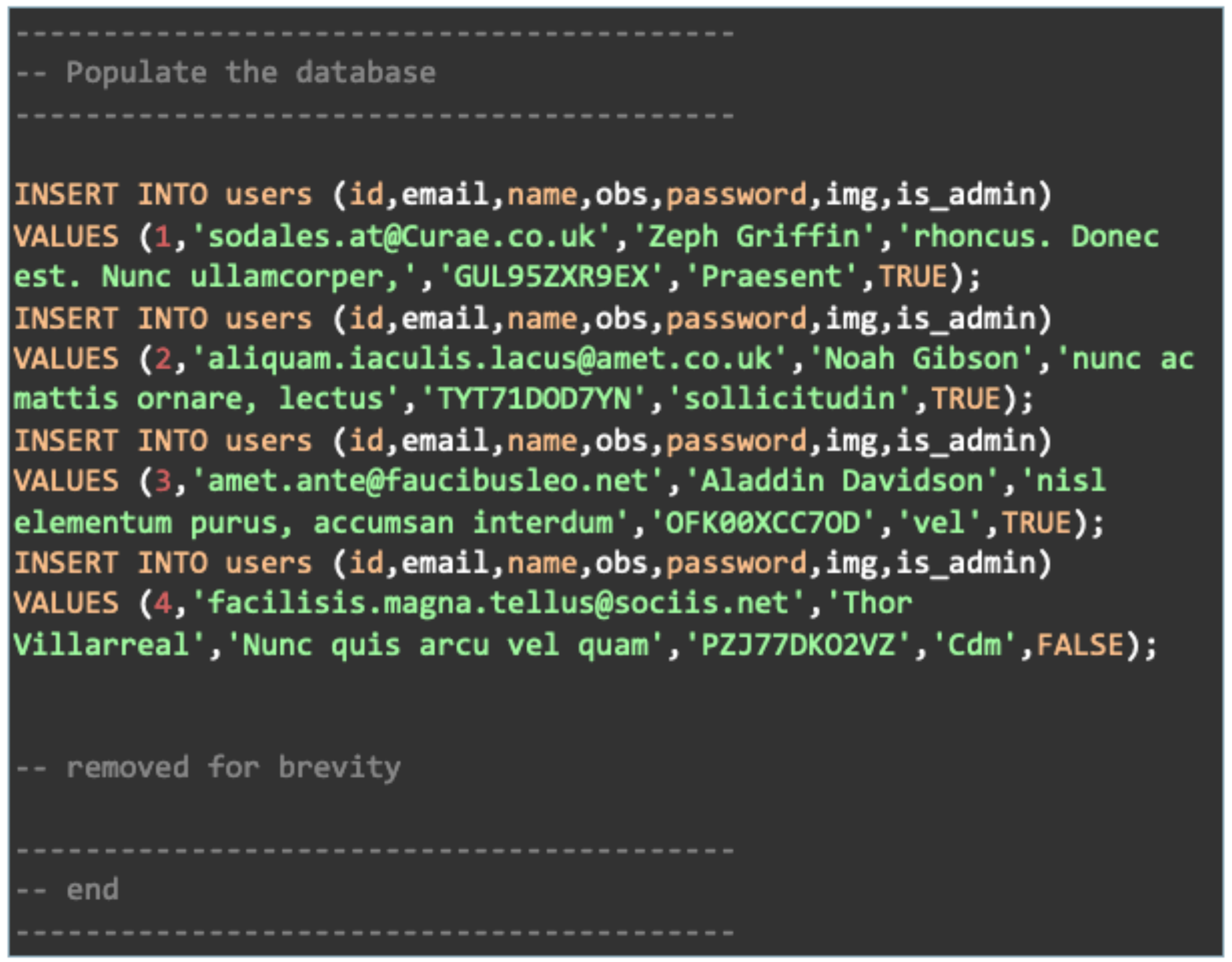
A6. MediaLibrary Database Population

A.1 Database schema



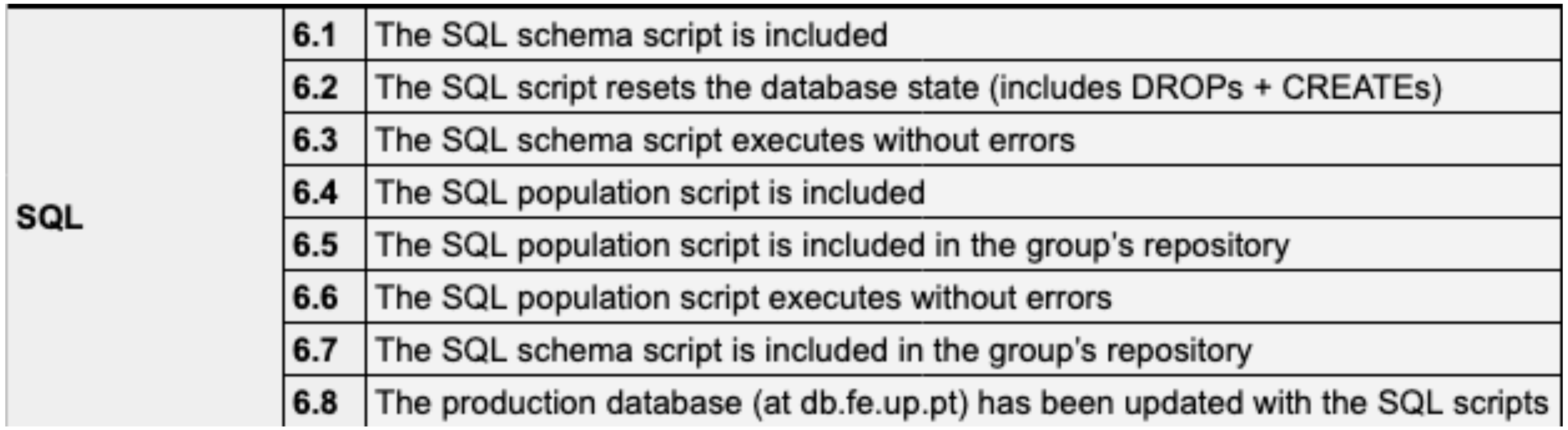
```
----------------------------------------
-- Drop old schema
----------------------------------------

DROP TABLE IF EXISTS wish_list CASCADE;
DROP TABLE IF EXISTS review CASCADE;
DROP TABLE IF EXISTS loan CASCADE;
DROP TABLE IF EXISTS item CASCADE;
DROP TABLE IF EXISTS nonbook CASCADE;
DROP TABLE IF EXISTS book CASCADE;
DROP TABLE IF EXISTS author_work CASCADE;
DROP TABLE IF EXISTS work CASCADE;
DROP TABLE IF EXISTS collection CASCADE;
DROP TABLE IF EXISTS author CASCADE;
DROP TABLE IF EXISTS location CASCADE;
DROP TABLE IF EXISTS publisher CASCADE;
DROP TABLE IF EXISTS users CASCADE;

DROP TYPE IF EXISTS media;


----------------------------------------
-- Types
----------------------------------------

CREATE TYPE media AS ENUM ('CD', 'DVD', 'VHS', 'Slides', 'Photos',
'MP3');

----------------------------------------
-- Tables
----------------------------------------

-- Note that a plural 'users' name was adopted because user is a
```

A.2 Database population



```
----------------------------------------
-- Populate the database
----------------------------------------

INSERT INTO users (id,email,name,obs,password,img,is_admin)
VALUES (1,'sodales.at@Curae.co.uk','Zeph Griffin','rhoncus. Donec
est. Nunc ullamcorper,','GUL95ZXR9EX','Praesent',TRUE);
INSERT INTO users (id,email,name,obs,password,img,is_admin)
VALUES (2,'aliquam.iaculis.lacus@amet.co.uk','Noah Gibson','nunc ac
mattis ornare, lectus','TYT71DOD7YN','sollicitudin',TRUE);
INSERT INTO users (id,email,name,obs,password,img,is_admin)
VALUES (3,'amet.ante@faucibusleo.net','Aladdin Davidson','nisl
elementum purus, accumsan interdum','OFK00XCC7OD','vel',TRUE);
INSERT INTO users (id,email,name,obs,password,img,is_admin)
VALUES (4,'facilisis.magna.tellus@sociis.net','Thor
Villarreal','Nunc quis arcu vel quam','PZJ77DKO2VZ','Cdm',FALSE);


-- removed for brevity

----------------------------------------
-- end
----------------------------------------
```

A6. Database Creation and Population Checklist



SQL	**6.1**	The SQL schema script is included
	6.2	The SQL script resets the database state (includes DROPs + CREATEs)
	6.3	The SQL schema script executes without errors
	6.4	The SQL population script is included
	6.5	The SQL population script is included in the group's repository
	6.6	The SQL population script executes without errors
	6.7	The SQL schema script is included in the group's repository
	6.8	The production database (at db.fe.up.pt) has been updated with the SQL scripts

PostgreSQL

Docker

- ➔ Docker is a key technology in LBAW (PostgreSQL, pgAdmin, Laravel)
- ➔ It is mandatory for deploying your prototypes and final products
- ➔ Docker is a lightweight virtualization environment, widely used to package applications and its dependencies in isolated containers.
- ➔ With Docker you can manage your product infrastructure as applications.
- ➔ Available for Windows, Mac and Linux - https://docs.docker.com/get-docker/
- ➔ Important: don't postpone using Docker.

PostgreSQL Docker Container

- ➔ Official PostgresSQL are available at: https://hub.docker.com/_/postgres
- ➔ Start a local PostgreSQL server with:
  - ➔
    ```
    docker run --name pgsql11 -e POSTGRES_PASSWORD=mysecretpassword -d postgres:11
    ```
- ➔ Run a local pgAdmin installation (available at localhost:80) with:
  - ➔
    ```
    docker run -p 80:80 \
    -e 'PGADMIN_DEFAULT_EMAIL=user@domain.com' \
    -e 'PGADMIN_DEFAULT_PASSWORD=SuperSecret' \
    -d dpage/pgadmin4
    ```

Docker Compose

- ➔ Docker Compose is used to setup multi-container Docker applications.
- ➔ A YAML file is used to configure the containers to start and run.
- ➔ The LBAW 'template-postgresql' repository, sets up two containers - https://git.fe.up.pt/lbaw/template-postgresql

```
version: '3'
services:

  postgres:
    image: postgres:11.13
    restart: always
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: pg!password
    ports:
      - "5432:5432"

  pgadmin:
    image: dpage/pgadmin4:6
    environment:
      PGADMIN_DEFAULT_EMAIL: postgres@lbaw.com
      PGADMIN_DEFAULT_PASSWORD: pg!password
    ports:
      - "4321:80"
    depends_on:
      - postgres
```

About the PostgreSQL Production Environment (**important!**) review

- ➔ A PostgreSQL database contains one or more schemas, which in turn contains one or more tables.
- ➔ All databases contain a public schema, which is used as default.
- ➔ In PostgreSQL's command line interface, you can view the current active schema with: `show search_path;`
- ➔ To change the schema for the current session use: `SET search_path TO <schema>;`
- ➔ **In the PostgreSQL setup at FEUP (db.fe.up.pt), the public schema is shared between all accounts**,
  - ➔ Tables created in the public schema are visible to all users (although not accessible).
    If you look at the tables in the publish schema, you will find a long list of tables.
  - ➔ **It is important to not use the public schema and instead create a schema with the name of your group (lbaw21gg)**.
- ➔ To create this schema, use the following command: `CREATE SCHEMA <lbaw21gg>;`
- ➔ To always use this schema as the default in your project, add the following line to the beginning of your SQL scripts.
  - ➔ `SET search_path TO <lbaw21gg>;`

References

Bibliography and Further Reading

- PostgreSQL Manual, Chapter 11. Indexes, https://www.postgresql.org/docs/current/indexes.html
- PostgreSQL Manual, Chapter 12. Full Text Search, www.postgresql.org/docs/current/textsearch.html
- Scott Ambler, The Object Primer, Cambridge University Press, 3rd Edition, 2004.
- UML — Metodologias e Ferramentas CASE (2ª Edição)
  Alberto Rodrigues da Silva, Carlos Videira, Centro Atlântico Editora, Maio 2005.
- Database Management Systems
  Raghu Ramakrishnan, Johannes Gehrke. McGRAW-Hill International Editions, 3rd Edition, 2003.
- Principles of Database Management
  Wilfried Lemahieu, Seppe Vanden Broucke, Bart Baesens. Cambridge University Press, 2018.

Lab Class #4

- ➔ Discuss the conceptual data model (A4)
- ➔ Develop and discuss the relational schema (A5)
  - ➔ Map the classes and relationships of the conceptual schema into relation schemas
  - ➔ For each relation, identify the functional dependencies (FD) that apply
  - ➔ Check if each relation is in BCNF
    - ➔ If the relation is not in BCNF and there are no other impediments, look for several possible decompositions (lossless)
    - ➔ If there is no satisfactory decomposition to BCNF and if the relation is no longer in 3NF, consider the decomposition lossless for 3NF, preserving the functional dependencies
  - ➔ Develop and test a first version of the database creation script in SQL

- ➔ Test the local development environment for PostgreSQL.
- ➔ Test the connection to the production PostgreSQL server at db.fe.up.pt