# PostgreSQL Tuning I: Indexing

## PostgreSQL

A powerful open-source relational database management system that is commonly used in enterprise-level applications.

## Database Tuning

The process of optimising a database's performance by adjusting its configuration parameters, such as indexing, caching and serialisability.

## Today's Focus

Indexing types and tuning

## Index

An index is a data structure that enables faster data retrieval by providing a quick lookup mechanism for frequently searched values.

An index can be created on one or more columns of a table, or even on partial rows on those columns.

An index can also add overhead to the database system, so it should be used with caution.

## Index Types

PostgreSQL provides 6 index types:

- B-tree
- Hash
- GiST
- SP-GiST
- GIN
- BRIN

By default, the `CREATE INDEX` command creates a B-tree index, which is suitable for most scenarios.

## Sequential Scan

Database system reads all the rows in a table, one row at a time, in a linear fashion, without using any indexes or sorting the data.
it is useful when the data needs to be read in a specific order or when most of the rows need to be accessed.

## Bitmap Heap Scan

Database system creates a bitmap for each value of a search key and then combining these bitmaps to identify the rows that match the search condition.
It can be faster than a sequential scan when querying large tables with selective criteria.

## Index Only Scan

Database system retrieves data from an index without the overhead accessing the data table. It requires that all the columns needed for the query are included in the index.

## Data Sets

1. sales_data:
   - 2 million rows
   - 14 columns: *region, country, item_type, sales_channel, order_priority, order_date, order_id, ship_date, units_sold, unit_price, unit_cost, total_revenue, total_cost, total_profits*

| region | country | item_type | … | total_revenue | total_cost | total_profits |
|---|---|---|---|---|---|---|
| Asia | Sri Lanka | Cosmetics | … | 3039414.4 | 1830670.16 | 1208744.24 |
| Europe | Norway | Baby Food | … | 1901836 | 1187679 | 714157 |
| Asia | Mongolia | Clothes | … | 19103.44 | 11275.32 | 7828.12 |
| Europe | France | Meat | … | 793518 | 477943.95 | 315574.05 |

sales_data

## Data Sets

2. geom_data:
   - 100 rows
   - point data

| p |
| --- |
| (102, 350) |
| (45, 78) |
| (403, 123) |
| (135, 238) |

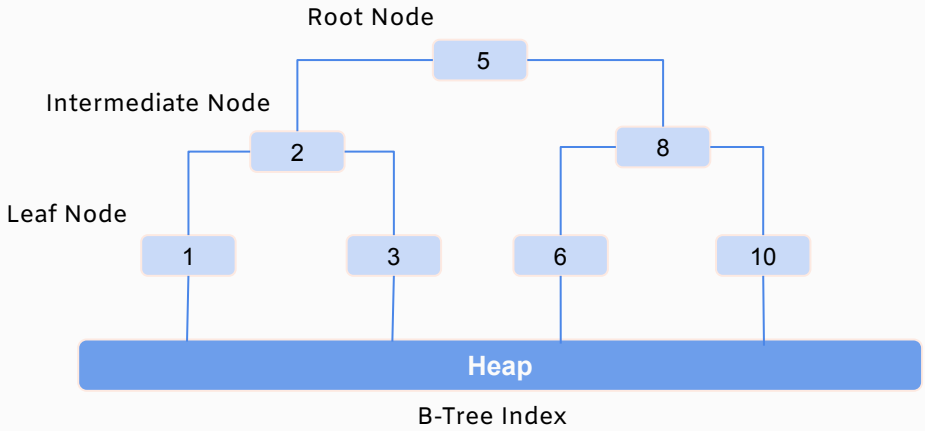
geom_data

## B-Tree Index

B-Tree index is a balanced tree data structure that stores the data in sorted order and has a search tree.

B-Tree index supports equality and range queries on data that can be sorted.

< <= = >= >




B-Tree Index

## B-Tree Index

```
CREATE INDEX btree_index
ON sales_data(order_id);
```

```
EXPLAIN ANALYZE SELECT *
FROM sales_data
WHERE order_id IN (198441684, 562844884, 838541638);
```

| Query Plan |
| --- |
| Index Scan using btree_index on sales_data |
| (cost=0.43..25.33 rows=3 width=107)<br>(actual time=0.089..0.122 rows=3 loops=1) |
| Planning Time: 0.425 ms |
| Execution Time: 0.238 ms |

| Equality Query | Query Method | Planning Time | Execution Time |
| --- | --- | --- | --- |
| Without B-tree index | Parallel Sequential Scan | 2.820 ms | 281.319 ms |
| With B-tree index | Index Scan | 0.425 ms | 0.238 ms |

## Hash Index

Hash index uses a collision-minimised hashing function and organises the indexed column with a 32-bit hash code.

Hash index only supports equality queries.

=

PostgreSQL creates a hash table that maps the indexed values to their corresponding table rows. With the great lookup performance of hash mapping, hash index is most suitable for simple equality-based queries.

## Hash Index

```
CREATE INDEX hash_order_id
ON sales_data
USING HASH(order_id);
```

```
EXPLAIN ANALYZE SELECT  *
FROM sales_data
WHERE order_id = 939825713;
```

| Query Plan |
| --- |
| Index Scan using hash_order_id on sales_data |
| (cost=0.00..8.02 rows=1 width=107)<br>(actual time=0.037..0.038 rows=1 loops=1) |
| Planning Time: 0.177 ms |
| Execution Time: 0.065 ms |

| Equality Query | Query Method | Planning Time | Execution Time |
| --- | --- | --- | --- |
| Without Hash index | Parallel Sequential Scan | 0.839 ms | 110.507 ms |
| With Hash index | Index Scan | 0.177 ms | 0.065 ms |

## GIN Index

Generalised Inverted Index (GIN) is designed to efficiently handle queries that involve composite values or multiple component values.

When a composite-typed column is GIN-indexed, the index contains a set of (key, posting list) pairs, of which the key is the element value and the posting list is the group of row IDs containing the key. When a query searches for one or more certain element values (i.e. keys), it can quickly locate the corresponding row IDs.

GIN supports many different user-defined indexing strategies and the following operators.

<@ @> = &&

```
CREATE EXTENSION pg_trgm;
CREATE INDEX gin_index
ON sales_data
USING gin(region gin_trgm_ops) ;
```

```
EXPLAIN ANALYZE SELECT COUNT( *)
FROM sales_data
WHERE region ILIKE "%Nor%");
```

| Query Plan |
|---|
| Parallel Bitmap Heap Scan on sales_data (cost=2814.09..42044.52 rows=126994 width=0) (actual time=32.398..180.586 rows=102024 loops=3) |
| Recheck Cond: ((region)::text ~~* '%Nor%'::text) |
| -> Bitmap Index Scan on gin_region (cost=0.00..2737.90 rows=304786 width=0) (actual time=29.995..29.996 rows=306071 loops=1) |
| Index Cond: ((region)::text ~~* '%Nor%'::text)<br>Planning Time: 0.229 ms Execution Time: 198.910 ms |

| Text Match Query | Query Method | Planning Time | Execution Time |
|---|---|---|---|
| Without GIN index | Parallel Sequential Scan | 0.553 ms | 246.327 ms |
| With GIN index | Parallel Bitmap Heap Scan + Bitmap Index Scan | 0.229 ms | 198.910 ms |

## BRIN Index

Block Range INdex(BRIN) stores information summary, such as the minimum and maximum value of each consecutive physical block range of a table.

The query planner uses bitmap index scan to look for the blocks whose summary matches the query conditions, and then it rechecks the records in the returned block and remove those that do not satisfy the conditions.

BRIN supports the following operators:

< <= = >= >

## Sort data table sales_data based on ship_date and create BRIN index on ship_date column

```
CREATE INDEX brin_index
ON sorted_sales_data
USING BRIN(ship_date);
```

```
EXPLAIN ANALYZE SELECT COUNT( *)
FROM sorted_sales_data
WHERE ship_date = '2010-01-01' AND ship_date <= '2010-01-08';
```

| Query Plan |
|---|
| Bitmap Heap Scan on sorted_sales_data (cost=12.03..18007.30 rows=1 width=0) (actual time=0.121..1.530 rows=10 loops=1) |
| Recheck Cond: ((ship_date <= '2010-01-08'::date) AND (ship_date = '2010-01-01'::date)) |
| -> Bitmap Index Scan on brin_index (cost=0.00..12.03 rows=7109 width=0) (actual time=0.109..0.109 rows=1280 loops=1) |
| Index Cond: ((ship_date <= '2010-01-08'::date) AND (ship_date = '2010-01-01'::date)) Planning Time: 0.322 ms Execution Time: 1.595 ms |

| Text Match Query | Query Method | Planning Time | Execution Time |
|---|---|---|---|
| Without BRIN index | Parallel Sequential Scan | 0.532 ms | 125.673 ms |
| With BRIN index | Parallel Bitmap Heap Scan + Bitmap Index Scan | 0.229 ms | 1.595 ms |

## GiST Index

GiST index stands for Generalised Search Tree index, is a tree-based structure that can handle a variety of different data types.

GiST can provide fast searching for complex data types such as geometric data and support nearest-neighbour searches.

GiST supports a wider range of operators, such as box operators and point operators.

<< &< &> >> <<| &<| |&> |>> @> <@ ~= &&

## GiST Index

```
CREATE INDEX gist_index
ON geom_data
USING gist(p);
```

```
EXPLAIN ANALYZE SELECT  *
FROM geom_data
ORDER BY p <-> point (120,450)  LIMIT 15;
```

| Query Plan |
| --- |
| Index Only Scan using gist_p on geom_data |
| (cost=0.14..10.14 rows=100 width=24)<br>(actual time=0.094..0.100 rows=15 loops=1) |
| Planning Time: 0.149 ms |
| Execution Time: 0.117 ms |

| Point Operator Query | Query Method | Planning Time | Execution Time |
| --- | --- | --- | --- |
| Without GiST index | Sequential Scan | 0.648 ms | 0.130 ms |
| With GiST index | Index Scan | 0.149 ms | 0.117 ms |

## SP-GiST Index

SP-GiST stands for space-partitioned GiST. It supports various non-balanced data structures compared to GiST.

SP-GiST supports a wider range of operators, such as box operators and point operators.

<< >> ~= <@ <<| |>>

The non-balanced structures divide the search space into partitions that may not have equal sizes, allowing for faster searches that match the partitioning rule.

## SP-GiST Index

```
CREATE INDEX spgist_index
ON geom_data
USING spgist(p);
```

```
EXPLAIN ANALYZE SELECT *
FROM geom_data
ORDER BY p <-> point '(101,450)' LIMIT 15;
```

| Query Plan |
| --- |
| Index Only Scan using spgist_index on geom_data |
| (cost=0.14..18.14 rows=100 width=24)<br>(actual time=0.035..0.040 rows=8 loops=1) |
| Planning Time: 0.171 ms |
| Execution Time: 0.060 ms |

| Point Operator Query | Query Method | Planning Time | Execution Time |
| --- | --- | --- | --- |
| Without SP-GiST index | Sequential Scan | 0.454 ms | 0.115 ms |
| With SP-GiST index | Index Scan | 0.171 ms | 0.060 ms |

## Multicolumn Indexes

In PostgreSQL, an index can be defined on multiple columns of a table.

In PostgreSQL, only B-Tree, GiST, GIN and BRIN supports multicolumn indexes. All multicolumn indexes can be used with queries on any subset of the indexed columns but the effectiveness is different for each index type based on column priorities.

| | B-Tree | Hash | GiST | SP-GiST | GIN | BRIN |
|---|---|---|---|---|---|---|
| **Multicolumn Index** | Support | Not support | Support | Not support | Support | Support |
| **Column Priority** | Leading columns | N.A. | First column | N.A. | Same priority | Same priority |

## A multicolumn index can be used for index scan

```
CREATE INDEX index_multi_btree
ON sales_data(ship_date, units_sold);
```

```
EXPLAIN ANALYZE SELECT COUNT( *)
FROM sales_data
WHERE ship_date = '2020-05-24'  AND units_sold >= 1000;
```

| **Query Plan** |
| --- |
| Bitmap Heap Scan on sales_data  (cost=13.31..1756.88 rows=476 width=107) (actual time=0.204..2.696 rows=478 loops=1) |
| Recheck Cond: ((ship_date = '2020-05-24'::date) AND (units_sold >= 1000)) |
| -> Bitmap Index Scan on index_multi_btree  (cost=0.00..13.19 rows=476 width=0) (actual time=0.136..0.136 rows=478 loops=1) |
| Index Cond: ((ship_date = '2020-05-24'::date) AND (units_sold >= 1000))<br>Planning Time: 0.271 ms Execution Time: 2.756 ms |

A multicolumn index can be used on only the leading column.

```
CREATE INDEX index_multi_btree
ON sales_data(ship_date, units_sold);
```

```
EXPLAIN ANALYZE SELECT *
FROM sales_data
WHERE ship_date = '2019-08-30';
```

| Query Plan |
| --- |
| Index Only Scan using index_multi_btree on sales_data |
| (cost=0.43..25.67 rows=528 width=0)<br>(actual time=0.087..0.186 rows=539 loops=1) |
| Planning Time: 0.044 ms |
| Execution Time: 0.249 ms |

The query planner would prefer a sequential scan over a index scan if the leading column is not in the query condition.

```
CREATE INDEX index_multi_btree
ON sales_data(ship_date, units_sold);
```

```
EXPLAIN ANALYZE SELECT *
FROM sales_data
WHERE units_sold > 5000;
```

| Query Plan |
| --- |
| Finalize Aggregate (cost=50649.25..50649.26 rows=1 width=8) (actual time=162.735..166.076 rows=1 loops=1) |
| Partial Aggregate (cost=49649.04..49649.05 rows=1 width=8) (actual time=158.853..158.853 rows=1 loops=3) |
| Parallel Seq Scan on sales_data (cost=0.00..48565.67 rows=433348 width=0) (actual time=0.068..135.158 rows=349525 loops=3) |
| Planning Time: 0.129 ms Execution Time: 166.101 ms |

## Unique Indexes

Unique indexes can be employed to ensure that the value of a column is unique or that the combination of values of multiple columns is unique.

In PostgreSQL, only B-tree index supports unique indexes.

Whenever a unique constraint or primary key is declared for a table in PostgreSQL, an index that enforces uniqueness is automatically created by the system.

```
CREATE UNIQUE INDEX unique_index_order_id
ON sales_data(order_id);
```

```
EXPLAIN ANALYZE SELECT *
FROM sales_data
WHERE order_id in (198441684, 562844884, 838541638) ;
```

| Query Plan |
| --- |
| Index Scan using unique_index_order_id on sales_data |
| (cost=0.43..25.33 rows=3 width=107)<br>(actual time=0.110..0.165 rows=3 loops=1) |
| Planning Time: 0.289 ms |
| Execution Time: 0.576 ms |

| Equality Query | Query Method | Planning Time | Execution Time |
| --- | --- | --- | --- |
| Without Unique index | Sequential Scan | 9.199 ms | 1159.194 ms |
| With Unique index | Index Scan | 0.289 ms | 0.576 ms |

```
CREATE UNIQUE INDEX unique_index_order_id
ON sales_data(order_id);
```

Insert a new row into the table with existing `order_id`. It will give an error message.

```
INSERT INTO sales_data(region, country, order_date, order_id)
VALUES('Europe', 'Luxembourg', '2019-08-30', 621165549)
```

| Error Message |
| --- |
| duplicate key value violates unique constraint "unique_index_order_id" |
| DETAIL: Key (order_id)=(621165549) already exists. |

## Partial Indexes

A partial index is a type of index created over a subset of a table, based on a specific condition expressed as a predicate.

All index types supports partial index.

Partial index reduces the size of the index, which will speed up the queries that uses the index.
Partial indexes are most suitable for data distributions that do not change.
In most cases, using a regular index is sufficient as the benefit of utilizing a partial index is very minimal.

```
CREATE INDEX partial_index_order_id
ON sales_data(order_id)
WHERE region = 'North America';
```

```
EXPLAIN ANALYZE SELECT *
FROM sales_data
WHERE region = 'North America' AND order_id = 160068500;
```

| Query Plan |
| --- |
| Index Scan using partial_index_order_id on sales_data |
| (cost=0.29..8.31 rows=1 width=107)<br>(actual time=0.206..0.208 rows=1 loops=1) |
| Planning Time: 1.469 ms |
| Execution Time: 0.251 ms |

| Equality Query | Query Method | Planning Time | Execution Time |
| --- | --- | --- | --- |
| Without Partial index | Sequential Scan | 2.748 ms | 130.504 ms |
| With Partial index | Index Scan | 0.289 ms | 0.576 ms |

```
CREATE INDEX partial_index_order_id
ON sales_data(order_id)
WHERE region = 'North America';
```

The following query cannot use this partial index because order 166568532 may not from region North America.

```
SELECT *
FROM sales_data
WHERE order_id = 166568532 ;
```

## Tip for using indexes

- There is no one-size-fits-all approach for determining which indexes to create or use. Experimentation and parameter tuning is often required to achieve satisfactory results.
- Always run `EXPLAIN ANALYZE` first. Running this command gathers data on the distribution of values in the table, which is used by the query planner to assign accurate costs to different query plans.
- Indexes may not be used all the time. The query planner chooses the optimal scan based on the estimated cost. If indexes are not used, it could mean the query can be completed more efficiently without using indexes or the indexes are indeed inappropriate.
- Test data does not equals to real data. Indexes are sensitive to data distribution. Hence, tuning indexes on test data may not lead to the same result on production data.

# PostgreSQL Tuning II: Serialisability

## Serialisability

A concept that ensures the consistency and correctness when executing a group of transactions concurrently in database.

It provides an illusion that those transactions are executed in sequence and do not interfere with each other, regardless of their actual order.

## Tuning on serialisability

The process of configuring the database to provide optimal performance and correctness guarantees for transactions that require serializability isolation level.

## Read Phenomena

Several transactions can run in parallel and result in incorrect data or data integrity issues.

PostgreSQL Documentation provides 4 types of read phenomena:

- Dirty read
- Non-repeatable read
- Phantoms
- Serialization anomaly

## Dirty Read

A transaction reads data written by concurrent uncommitted transactions.

## Non-repeatable Read

A transaction re-reads data and finds that data has been modified by another transaction.

## Phantoms

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

## Serialization anomaly

The result of committing a series of transactions is not equivalent with committing these transactions in order.

## ACID Properties

4 key properties that define a transaction: Atomicity, Consistency, *Isolation*, and Durability.

## Isolation Levels

The ANSI standard defines 4 isolation levels:

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

## Read Uncommitted

Transactions can see uncommitted changes made by other concurrent transactions. Lowest isolation level.

## Read Committed

Transactions can only see committed changes made by other concurrent transactions.

## Repeatable Read

Transactions will always see the same data throughout execution, even other transactions modify it.

## Serializable

Concurrent transactions execute as if they were executing serially.

## Isolation Level in PostgreSQL

The relationship between read phenomena and isolation levels in PostgreSQL is given below

| | Dirty Read | Non-repeatable Read | Phantoms | Serializability 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 |

## Test Table

Warehouse
- A simple table to demonstrate the consequences of 4 isolation levels in PostgreSQL

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |
| 3 | Caruso | 20 | 3000 | Indonesia |

Employee Data

## Transaction

We use 2 transactions denoted T1 and T2.
Generally, we perform write operations(INSERT, DELETE, UPDATE) in T1 and SELECT queries in T2.

| T1 |
| --- |
| INSERT…<br>UPDATE…<br>DELETE… |

| T2 |
| --- |
| SELECT…<br><br>SELECT… |

## 1.Read Uncommitted Level - Dirty Read

| T1 | T2 |
| --- | --- |
| BEGIN;<br>SET TRANSACTION ISOLATION LEVEL<br>READ UNCOMMITTED; | BEGIN;<br>SET TRANSACTION ISOLATION LEVEL<br>READ UNCOMMITTED; |
| | SELECT * FROM warehouse<br>WHERE w_id = 1;(Table 1) |
| UPDATE warehouse<br>SET w_name = 'Johnson'<br>WHERE w_id = 1; | |
| | SELECT * FROM warehouse<br>WHERE w_id = 1;(Table 2) |

## 1.Read Uncommitted Level - Dirty Read

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |

Table 1

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |

Table 2

It is unexpected not to see the change on record where w_id = 1 in transaction 2, since we are using ‘READ UNCOMMITTED’ level.

Reason: Internally only 3 distinct isolation levels are implemented, and PostgreSQL's Read Uncommitted mode behaves like Read Committed.

## 2.Read Committed Level - Dirty Read

Behaves the same as Read Uncommitted level.

## 2.Read Committed Level - Non-repeatable Read

| T1 | T2 |
| --- | --- |
| BEGIN;<br>SET TRANSACTION ISOLATION LEVEL READ COMMITTED; | BEGIN;<br>SET TRANSACTION ISOLATION LEVEL READ COMMITTED; |
| | SELECT * FROM warehouse<br>WHERE w_id = 1;(Table 1) |
| UPDATE warehouse<br>SET w_salary = w_salary + 1000<br>WHERE w_id = 1;<br>COMMIT; | |
| | SELECT * FROM warehouse<br>WHERE w_id = 1; (Table 2) |

## 2.Read Committed Level - Non-repeatable Read

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |

Table 1

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 3000 | Singapore |

Table 2

Here we can observe an undesirable phenomenon. The salary of record 1 read by T2 changed from 2000 to 3000 within the transaction.

READ COMMITTED level cannot prevent Non-repeatable Read.

## 2.Read Committed Level - Phantoms

| T1 | T2 |
| --- | --- |
| BEGIN;<br>SET TRANSACTION ISOLATION LEVEL READ COMMITTED; | BEGIN;<br>SET TRANSACTION ISOLATION LEVEL READ COMMITTED; |
| | SELECT * FROM warehouse WHERE w_age > 21;(Table 1) |
| INSERT INTO warehouse VALUES(4,'Daniel',25,4000,'Singapore');<br>COMMIT; | |
| | SELECT * FROM warehouse WHERE w_age > 21; (Table 2) |

## 2.Read Committed Level - Phantoms

| w_id | country | w_ag | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |

Table 1

| w_id | country | w_ag | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |
| 4 | Daniel | 25 | 4000 | Singapore |

Table 2

## 2.Read Committed Level - Phantoms

T2 re-executes a query under a search condition(w_age>21) and finds that the results have changed due to another recently-committed transaction(T1 inserted a new record).

READ COMMITTED level cannot prevent Phantoms too.

| | Dirty read | Non-repeatable Read | Phantoms | Serialization anomaly |
|---|---|---|---|---|
| **Read Uncommitted** | Allowed, but not in PG | Possible | Possible | Possible |
| **Read Committed** | Not possible | Possible | Possible | Possible |

## 3.Repeatable Read Level - Non-repeatable Read

| T1 | T2 |
| --- | --- |
| BEGIN;<br>SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; | BEGIN;<br>SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; |
| | SELECT * FROM warehouse<br>WHERE w_id = 1;(Table 1) |
| UPDATE warehouse<br>SET w_salary = w_salary + 1000<br>WHERE w_id = 1;<br>COMMIT; | |
| | SELECT * FROM warehouse<br>WHERE w_id = 1; (Table 2) |

## 3.Repeatable Read Level - Non-repeatable Read

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |

Table 1

| w_id | country | w_age | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |

Table 2

REPEATABLE READ level can prevent Non-repeatable Read.

## 3.Repeatable Read Level - Non-repeatable Read

| T1 |
| --- |
| *Continue last page… |
| |

| T2 |
| --- |
| *Continue last page… |
| UPDATE warehouse<br>SET w_salary = w_salary + 1000<br>WHERE w_id = 1;(?) |

```
Error: could not serialize access
due to concurrent update
```

Moreover, we cannot update that record in T2 before T2 commits.

## 3.Repeatable Read Level - Phantoms

| T1 | T2 |
| --- | --- |
| BEGIN;<br>SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; | BEGIN;<br>SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; |
| | SELECT * FROM warehouse WHERE w_age > 21;(Table 1) |
| INSERT INTO warehouse VALUES(4,'Daniel',25,4000,'Singapore');<br>COMMIT; | |
| | SELECT * FROM warehouse WHERE w_age > 21; (Table 2) |

## 3.Repeatable Read Level - Phantoms

| w_id | country | w_ag | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |

Table 1

| w_id | country | w_ag | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |

Table 2

## 3.Repeatable Read Level - Phantoms

T2 re-executes a query under a search condition(w_age>21) and finds that the results is the same as the previous one, although T1 inserted a new record.

REPEATABLE READ level can prevent Phantoms.

| | Dirty read | Non-repeatable Read | Phantoms | Serialization anomaly |
|---|---|---|---|---|
| **Repeatable Read** | Not possible | Not possible | Allowed, but not in PG | Possible |

## 3.Repeatable Read Level - Serialisability

**T1**

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ;

INSERT INTO warehouse
(w_name,w_age,w_salary,w_country)
SELECT 'sum',0,SUM(w_salary),'All'
FROM warehouse;
COMMIT;
```

**T2**

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ;

INSERT INTO warehouse
(w_name,w_age,w_salary,w_country)
SELECT 'sum',0,SUM(w_salary),'All'
FROM warehouse;
COMMIT;
```

```
SELECT * FROM warehouse;
```
(Table 1)

## 3.Repeatable Read Level - Serialisability

| w_id | country | w_ag | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |
| 3 | Caruso | 20 | 3000 | Indonesia |
| 4 | sum | 0 | 6000 | All |
| 5 | sum | 0 | 6000 | All |

Table 1

The result is not equivalent to the result when T1 and T2 are executed in order.

REPEATABLE READ level cannot prevent Serialisability.

## 3.Repeatable Read Level - Serialisability

| | Dirty read | Non-repeatable Read | Phantoms | Serialization anomaly |
|---|---|---|---|---|
| **Repeatable Read** | Not possible | Not possible | Allowed, but not in PG | Possible |

## 4.Serialisable - Serialisability

**T1**

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ;

INSERT INTO warehouse
(w_name,w_age,w_salary,w_country)
SELECT 'sum',0,SUM(w_salary),'All'
FROM warehouse;
COMMIT;
```

**T2**

```
BEGIN;
SET TRANSACTION ISOLATION LEVEL
REPEATABLE READ;

INSERT INTO warehouse
(w_name,w_age,w_salary,w_country)
SELECT 'sum',0,SUM(w_salary),'All'
FROM warehouse;
COMMIT;
```

## 4.Serialisable - Serialisability

After T1 commits, Serialisable level prevents T2 from committing:

ERROR: could not serialize access due to read/write dependencies among transactions
DETAIL: Reason code: Canceled on identification as a pivot, during commit attempt.
HINT: The transaction might succeed if retried.

| | Dirty read | Non-repeatable Read | Phantoms | Serialization anomaly |
|---|---|---|---|---|
| **Serializable** | Not possible | Not possible | Not possible | Not possible |

REPEATABLE READ level cannot prevent Serialisability.

## 4.Serialisable - Serialisability

We rollback T2 and retry:

| w_id | country | w_ag | w_salary | w_country |
|---|---|---|---|---|
| 1 | Alice | 22 | 2000 | Singapore |
| 2 | Bob | 24 | 1000 | Indonesia |
| 3 | Caruso | 20 | 3000 | Indonesia |
| 4 | sum | 0 | 6000 | All |
| 5 | sum | 0 | 12000 | All |

Table 1

REPEATABLE READ level cannot prevent Serialisability.

## transaction_isolation

This parameter reflects the current transaction's isolation level.

```
SHOW TRANSACTION ISOLATION LEVEL
( or: SELECT current_setting('transaction_isolation');)

 current_setting
-----------------
 read committed
```

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

## default_transaction_isolation

At the beginning of each transaction, the current transaction's isolation level is set to the current value of default_transaction_isolation.
Its default value in PostgreSQL is 'read_committed'.

User can change default_transaction_isolation with the following command:

```
ALTER DATABASE <db name>
SET DEFAULT_TRANSACTION_ISOLATION TO 'read committed';
```

## Tuning on Serialisability

The default transaction_isolation level in PostgreSQL is ‘read_committed’ in order to provide a reasonable balance between data consistency and concurrency.

Higher isolation levels reduce possibilities of concurrency issues such as deadlocks, while it may also result in reduced concurrency and increased transaction overhead too, which reduces the performance.

Choosing isolation level should be based on the specific requirements of the application and workload.