PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Laporan Tugas Besar

Mini Database Management System Development

Oleh:

SUPER GROUP 7

PostgreZQL

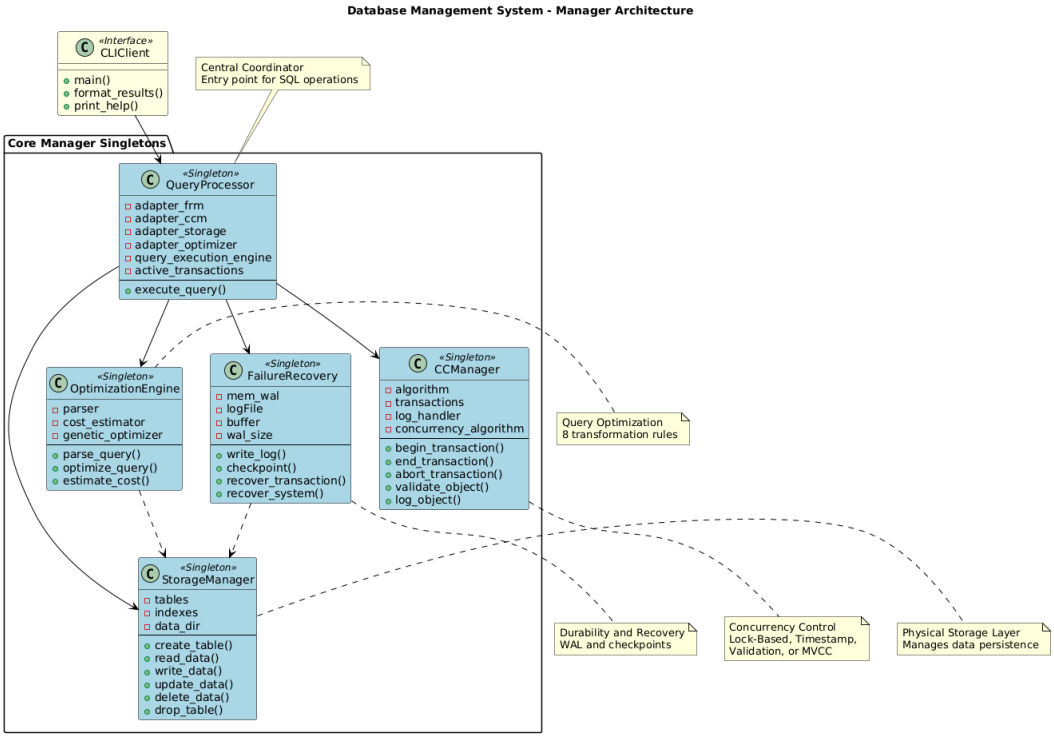
Part I. Gambaran Umum DBMS

Database Management System (DBMS) adalah sebuah sistem perangkat lunak yang didesain untuk menangani sekumpulan data yang saling berkaitan serta menyediakan alat untuk mengakses, mengelola, dan memanipulasi data tersebut. DBMS juga memastikan integritas dan sekuritas data, serta mendukung akses konkuren oleh *multiple users.* DBMS umumnya digunakan dalam pengelolaan data-data penting dalam organisasi, pembuatan *data storage*, penyediaan mekanisme untuk operasi data, dan jaminan keamanan data bahkan untuk kasus *crash* hingga percobaan akses tidak terotorisasi.

Mini Database Management System (mDBMS), seperti yang telah dijelaskan pada *Mega Project Specification,* adalah versi simplifikasi dari DBMS. Proyek ini berfokus untuk membangun komponen kunci dari sebuah DBMS. Komponen-komponen yang terdapat pada mDBMS PostgreZQL yang telah diimplementasi merupakan Query Processor (QP), Query Optimizer (QO), Failure Recovery Manager (FRM), Concurrency Control Manager (CCM), dan Storage Manager (SM). Sebuah Query dibaca dan di-*parsing* menjadi sebuah pohon instruksi oleh QO. Instruksi tersebut dibaca oleh QP dan dijalankan setiap instruksi berdasarkan pohonnya menuju SM, dengan setiap perjalanan instruksi tersebut diterapkan logging melalui FRM dan validasi *lock* oleh CCM. FRM melakukan pencatatan untuk setiap operasi yang terjadi untuk setiap transaksi, dan bertanggung jawab memberikan operasi-operasi pada QP jika terjadi Rollback. CCM mengatur distribusi terhadap kepemilikan *lock* untuk setiap transaksi dan mencegah terjadinya sebuah *deadlock* dengan granularitas tabel. SM merupakan tempat penyimpanan data sesungguhnya dari basis data, dan menyediakan abstraksi kepada QP untuk akses baca dan tulis terhadap data tersebut.

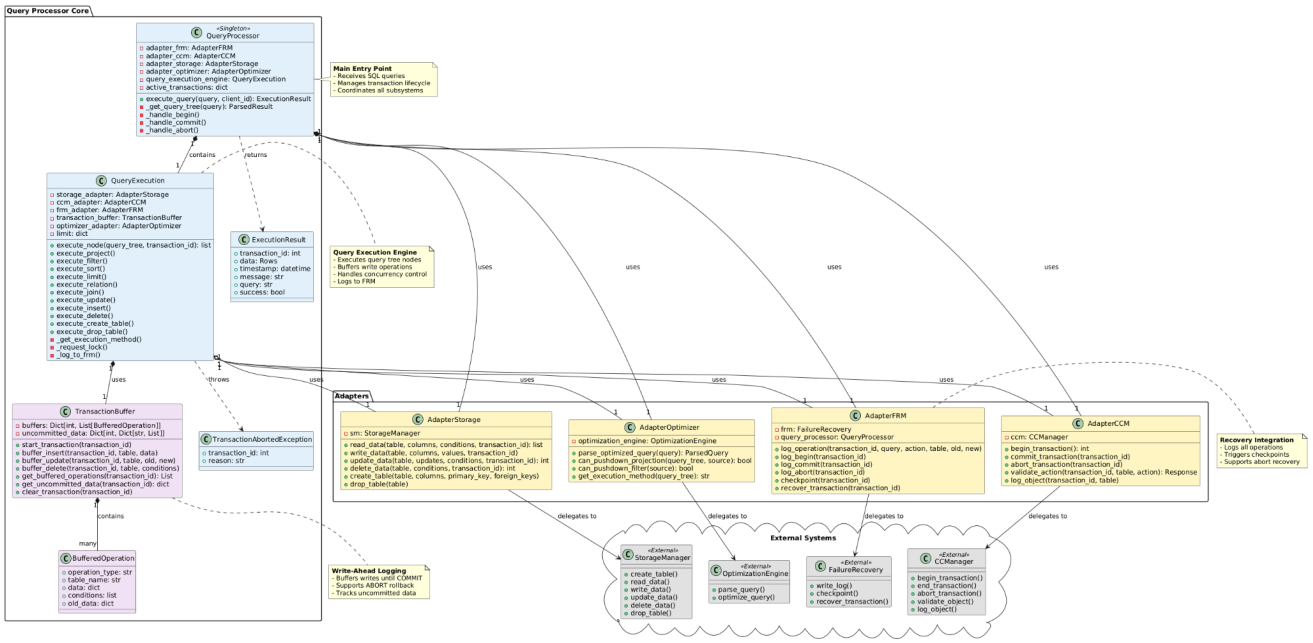
Pada implementasi ini, mDBMS melayani *multiple user* melalui arsitektur *Client-Server*, menyediakan sebuah *endpoint* untuk mengirim *request* dan menerima *response*. Sesuai

dengan *scope* proyek ini, akses hanya dapat dilakukan melalui *localhost*, dengan setiap user terhubung pada setiap terminal.



Part II. Detail Komponen DBMS

1. Query Processor



Query Processor berperan sebagai koordinator utama terhadap komponen-komponen lainnya dalam proses eksekusi query dan transaksi. Operasi-operasi dari sebuah query dijalankan sesuai plan yang diberikan Query Optimizer terhadap Storage Manager dan diawasi oleh Concurrency Control Manager dan Failure Recovery Manager.

- **QueryProcessor**

  Kelas QueryProcessor merupakan kelas *singleton* utama yang merepresentasikan komponen Query Processor. Kelas QueryProcessor berperan sebagai inti dari mDBMS. Kelas ini memiliki akses terhadap kelas-kelas komponen lainnya, dan fungsionalitasnya bertumpu pada fungsi *execute_query()*. QueryProcessor bertugas melakukan manajemen terhadap *lifecycle* transaksi-transaksi yang melewatinya, mengurusi BEGIN TRANSACTION dan COMMIT. Menempelkan transaction_id baru terhadap *request* transaksi baru untuk sebuah klien, dan juga melapisi *single query* dengan transaksi. QueryProcessor juga berperan sebagai

penerus instruksi-instruksi *recovery* dari Failure Recovery Management menuju Storage Management ketika terjadi *Rollback* dari transaksi atau *system failure*.

- **TransactionBuffer**

  Untuk setiap operasi WRITE yang dilakukan oleh sebuah transaksi, data tersebut tidak langsung dimasukkan ke dalam *storage manager*, sedangkan ditampung terlebih dahulu dalam *buffer*. Hal ini memastikan level konsistensi mencapai COMMITTED READ, yaitu setiap transaksi hanya dapat membaca data yang telah di-commit. Pada saat COMMIT, baru melakukan pengosongan isi buffer dan penyimpanan data ke dalam Storage Manager. Pendekatan tersebut dinamakan Write-Ahead-Log. Terdapat batasan pada implementasi ini untuk menangani buffer yang terlalu penuh, yaitu *checkpoint*. Modifikasi data direpresentasikan dalam bentuk [old_data, new_data] untuk setiap transaksi. Ini merupakan pilihan desain yang penting agar transaksi membaca dan menulis new_data yang menggantikan old_data hanya pada transaksi sendiri sebelum commit, memenuhi prinsip *Isolation*.

- **QueryExecution**

  Kelas QueryExecution merupakan engine utama yang melakukan eksekusi query tree secara rekursif untuk menghasilkan hasil query yang diinginkan. Ini merupakan kelas yang bertanggung jawab menerjemahkan struktur query tree hasil optimisasi menjadi serangkaian operasi konkrit terhadap data. QueryExecution mengimplementasikan berbagai operator relasional standar seperti PROJECT untuk seleksi kolom, FILTER untuk kondisi WHERE, JOIN untuk menggabungkan tabel, SORT untuk pengurutan, dan LIMIT untuk pembatasan hasil. Setiap operator diimplementasikan dengan strategi yang berbeda tergantung pada kompleksitas query dan kemampuan komponen di bawahnya.

  Engine ini dirancang dengan prinsip pushdown optimization, di mana operasi yang sederhana dan dapat ditangani langsung oleh Storage Manager akan diteruskan ke layer tersebut untuk efisiensi eksekusi. Ketika source query adalah

RELATION sederhana, operasi PROJECT dan FILTER dapat di-*pushdown* langsung ke Storage Manager sehingga mengurangi transfer data dari storage ke memory. Sebaliknya, untuk query yang kompleks seperti subquery, JOIN, atau kondisi yang mengandung OR/NOT, eksekusi dilakukan di memori oleh Query Processor sendiri setelah mengambil data dari storage. Pendekatan hybrid ini memastikan balance antara performa dan kemampuan menangani query kompleks.

Untuk operasi DML seperti INSERT, UPDATE, dan DELETE, QueryExecution mengimplementasikan pola Read-Modify-Write yang terintegrasi dengan TransactionBuffer. Setiap modifikasi yang telah dilakukan terhadap sekumpulan row akan ditampung pada TransactionBuffer dan operasi-operasi *query* akan memprioritaskan keberadaan data yang terdapat pada buffer. Ketika sebuah transaksi melakukan UPDATE, sistem terlebih dahulu membaca row yang akan dimodifikasi dengan menerapkan buffered operations agar transaksi dapat melihat hasil modifikasinya sendiri. Kemudian, sistem mengevaluasi expression pada *SET* clause yang dapat berupa ekspresi aritmatika kompleks seperti *SET salary = salary * 1.1 + bonus*. Hasil evaluasi ini tidak langsung ditulis ke storage, melainkan di-buffer terlebih dahulu dalam TransactionBuffer hingga COMMIT dieksekusi.

QueryExecution juga bertanggung jawab mengkoordinasikan validasi *concurrency control* untuk setiap akses tabel. Sebelum melakukan READ atau WRITE terhadap sebuah tabel, sistem memanggil method *_validate_with_retry()* yang berkomunikasi dengan Concurrency Control Manager untuk memastikan transaksi memiliki hak akses yang sesuai. Jika lock tidak dapat diperoleh segera, sistem akan menunggu dengan retry logic hingga timeout tertentu. Jika transaksi di-abort oleh CCM karena deadlock atau conflict, QueryExecution akan melempar exception TransactionAbortedException yang kemudian ditangani oleh QueryProcessor untuk melakukan recovery melalui Failure Recovery Manager.

Integrasi terhadap komponen lain dilakukan melalui *design pattern* adapter untuk setiap komponen, agar metode-metode komponen diproses terlebih dahulu sesuai keperluan Query Processor dalam menjalankan sebuah *query*.

a. Integrasi dengan Query Optimizer

Integrasi Query Processor dengan komponen Query Optimizer bekerja melalui kelas adapter bernama AdapterOptimizer. Adapter ini akan menyimpan kelas OptimizationEngine milik Query Optimizer serta berperan dalam menjembatani proses *parsing* terhadap *query*, mengecek apakah pushdown dapat dilakukan (terhadap project, filter), menentukan metode eksekusi pada suatu *node* dalam QueryTree, serta mengecek apakah suatu kondisi filter memerlukan pemrosesan dalam memori (*in-memory filtering*). Secara umum, metode-metode di atas bekerja dengan memanggil *method* yang dimiliki oleh OptimizationEngine atau dengan memanfaatkan struktur dan isi kelas QueryTree. Implementasi kelas AdapterOptimization tertera di bawah ini.

```
class AdapterOptimizer:
    def __init__(self):
        self.optimization_engine = OptimizationEngine()

    def parse_optimized_query(self, query: str) -> ParsedQuery:
        parsed_query = self.optimization_engine.parse_query(query)
        optimized_query = self.optimization_engine.optimize_query(parsed_query)
        return optimized_query

    def can_pushdown_projection(self, query_tree: QueryTree, source: QueryTree) ->
bool:
        return source.type == "RELATION"

    def can_pushdown_filter(self, source: QueryTree) -> bool:
        return source.type == "RELATION"

    def get_execution_method(self, query_tree: QueryTree) -> str:
        node_type = query_tree.type
        if hasattr(query_tree, 'method') and query_tree.method:
            # Use method specified in the tree
            return query_tree.method

        # Default methods based on node type
        if node_type in ["RELATION", "SELECT"]:
            return "sequential_search"
```

```
        elif node_type == "JOIN":
            return "nested_loop"
        else:
            return ""

    def should_use_in_memory_filtering(self, condition_types: list) -> bool:
        complex_conditions = {
            "IN_EXPR", "NOT_IN_EXPR",
            "EXISTS_EXPR", "NOT_EXISTS_EXPR",
            "BETWEEN_EXPR", "NOT_BETWEEN_EXPR",
            "OPERATOR"  # AND/OR/NOT may contain complex nested conditions
        }

        return any(ct in complex_conditions for ct in condition_types)

def print_query_tree(query_tree, indent=0):
    if query_tree is None:
        return

    prefix = "  " * indent
    node_info = f"{query_tree.type}"

    if query_tree.val:
        node_info += f" [{query_tree.val}]"

    print(f"{prefix}{node_info}")

    for child in query_tree.childs:
        print_query_tree(child, indent + 1)
```

Penggunaan utama dari adapter ini terletak pada kelas utama QueryProcessor, juga pada kelas QueryExecution. Beberapa contoh penggunaan adapter adalah sebagai berikut.

```
#1 kelas QueryProcessor
class QueryProcessor:
    ...

    def __init__(self):
        ...
        self.adapter_optimizer = AdapterOptimizer() # <-- Adapter sebagai atribut
        ...

#2 Private method pada kelas QueryProcessor
     # ---------------------------- private methods -----------------------------
    def _get_query_tree(self, query:str):
        return self.adapter_optimizer.parse_optimized_query(query)# <-- Adapter
```

```
--------------------------------------------------------------------------------------

#3 kelas QueryExecution
class QueryExecution:
    def __init__(self, storage_adapter=None, ccm_adapter=None, storage_manager=None,
frm_adapter=None):
        ...

        # Initialize optimizer adapter for optimization decisions
        self.optimizer_adapter = AdapterOptimizer() # <-- Adapter sebagai atribut

        ...

# for select and join methods
    def _get_execution_method(self, node: QueryTree) -> str:
        """Delegate to optimizer adapter for execution method decisions"""
        return self.optimizer_adapter.get_execution_method(node) # <-- Adapter
```

b. Integrasi dengan Failure Recovery Manager

Integrasi Query Processor dengan komponen Failure Recovery Manager bekerja melalui kelas adapter bernama AdapterFRM. Adapter ini akan menyimpan kelas FailureRecovery milik komponen Failure Recovery Manager serta berperan dalam menjembatani proses *logging, commit, abort, recovery* transaksi, dan hal-hal lain yang berkaitan dan menunjang proses tersebut. Hal ini mencakup memulai *logging*, membuat *log* terkait mulainya transaksi, *log* terkait penulisan ke *disk*, *commit/abort* transaksi, serta inisiasi *recovery* dan perolehan informasi terkait (*undo list*, *checkpointint,* dsb.). Spesifikasi singkat kelas AdapterFRM adalah sebagai berikut.

```
class AdapterFRM:
    def __init__(self, wal_size=50):
        """
        Initialize adapter with FRM singleton instance

        Args:
            wal_size (int): Size of write-ahead log before flushing to disk
        """
        self.frm = FailureRecovery(wal_size=wal_size)
        self.query_processor = None  # Will be set by QueryProcessor
```

```
    def log_operation(self, transaction_id: int, query: str, action: str,
                      table_name: str, old_data=None, new_data=None) -> None:
        ...

    def log_begin(self, transaction_id: int) -> None:
        """Log transaction start"""
        ...

    def log_write(self, transaction_id: int, query: str, table_name: str,
                  old_data=None, new_data=None) -> None:
        """Log a write operation (INSERT/UPDATE/DELETE)"""
        ...

    def _flush_to_disk(self) -> None:
        """Force immediate flush of WAL to disk"""
        ...

    def log_commit(self, transaction_id: int) -> None:
        """Log transaction commit - this flushes WAL to disk"""
        ...

    def log_abort(self, transaction_id: int) -> None:
        """Log transaction abort"""
        ...

    def recover_transaction(self, transaction_id: int) -> None:
        """
        Recover (undo) a transaction by reverting all changes that were flushed to
storage.
        This is called when a transaction aborts after checkpoint has flushed some
operations.

        Args:
            transaction_id: ID of transaction to recover
        """
        ...

    def get_undo_list(self):
        """Get list of transactions that need to be undone"""
        ...

    def recover(self, criteria=None):
        """Trigger recovery process"""
        ...

    def recover_system_crash(self) -> None:
        """
        Recover from system crash on startup.
        Performs REDO for committed transactions after last checkpoint.
```

```
        Performs UNDO for uncommitted transactions.
        """
        ...

    def _create_checkpoint(self) -> None:
        """Create a checkpoint: flush all buffered transactions and write checkpoint
log"""
        ...

    def _commit_all_buffered_transactions(self) -> None:
        """Commit all transactions that have buffered operations to storage"""
        ...
```

c. Integrasi dengan Storage Manager

Integrasi Query Processor dengan komponen Storage Manager bekerja melalui kelas adapter bernama AdapterStorage. Adapter ini akan menyimpan kelas StorageManager milik komponen Storage Manager serta berperan dalam menjembatani proses *data retrieval*, *data write, data deletion*, *data update*, dan hal-hal lain yang berkaitan dan menunjang proses tersebut sehingga komponen di atasnya cukup berinteraksi melalui adapter ini. Spesifikasi singkat kelas AdapterStorage adalah sebagai berikut:

```
class AdapterStorage:
    """
    Adapter for Storage Manager operations.
    Provides a clean interface between query execution and storage layer.
    """

    def __init__(self, storage_manager=None, data_dir=None):
        """Initialize with either an existing storage manager or create a new one"""
        if storage_manager:
            self.sm = storage_manager
        else:
            self.sm = StorageManager(data_dir=data_dir) if data_dir else
StorageManager()

    def read_data(self, table_name, columns=None, conditions=None,
transaction_id=None):
        """
        Read data from storage with optional projection and filtering.
        """

    def write_data(self, table_name, columns, values, conditions=None,
transaction_id=None):
```

```
        """
        Write data to storage (INSERT if conditions empty, UPDATE if conditions
present).
        """

    def delete_data(self, table_name, conditions=None, transaction_id=None):
        """
        Delete data from storage.
        """

    def drop_table(self, table_name):
        """
        Drop a table from storage.
        """

    def batch_update_data(self, table_name, old_data_list, new_data_list,
transaction_id=None):
        """
        Batch update data using old/new data matching (optimized for FRM/transactions).

        This method uses the storage manager's update_by_old_new_data which:
        1. Matches rows by PRIMARY KEY first (efficient)
        2. Falls back to exact match if no PK match
        3. Performs all updates in one operation
        """

    # Legacy methods for backward compatibility
    def storage_select(self, table_name, conds=None, projections=None):
        """Legacy method - use read_data instead"""

    def storage_insert(self, table_name, row_dict):
        """Legacy method - use write_data instead"""

    def storage_update(self, table_name, set_dict, condition_tuple):
        """Legacy method - use write_data instead"""

    def storage_delete(self, table_name, condition_tuple):
        """Legacy method - use delete_data instead"""
```

d. Integrasi dengan Control Concurrency Manager

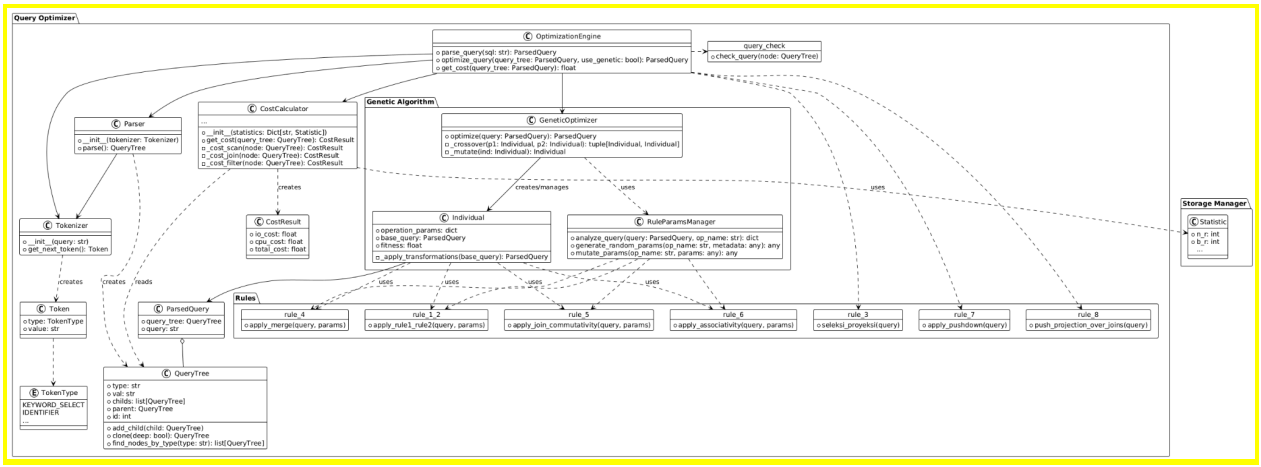
Integrasi Query Processor dengan komponen Concurrency Control Manager bekerja melalui kelas adapter bernama AdapterCCM. Adapter ini akan menyimpan kelas ConcurrencyControlManager milik komponen Concurrency Control Manager. Adapter ini akan menjembatani Query Processor untuk meminta beberapa proses ke Concurrency Control Manager, seperti memulai

transaksi, mengakhiri transaksi (commit), membatalkan transaksi (abort), meminta log, dan memvalidasi aksi yang dapat dilakukan. Spesifikasi singkat kelas AdapterCCM adalah sebagai berikut.

```
class AdapterCCM:

    def __init__(self, algorithm: AlgorithmType):
        """
        Inisialisasi adapter dan instance CCManager internal
        berdasarkan algoritma yang dipilih.


        """
        self.ccm = CCManager(algorithm=algorithm)

    def begin_transaction(self) -> int:
        """
        Memulai transaksi baru via CCM dan mengembalikan ID transaksi.
        """
        return self.ccm.begin_transaction()

    def commit_transaction(self, transaction_id: int):
        """
        Menjalankan proses commit untuk transaksi via CCM.
        Memanggil 'end_transaction' pada CCManager.
        """
        self.ccm.end_transaction(transaction_id)

    def abort_transaction(self, transaction_id: int):
        """
        Menjalankan proses abort untuk transaksi via CCM.
        Memanggil 'abort_transaction' pada CCManager.
        """
        self.ccm.abort_transaction(transaction_id)

    def validate_action(self, transaction_id: int, table_name: str, action_type: str)
-> Response:
        """
        Memvalidasi apakah suatu aksi (read/write) diizinkan pada sebuah objek (tabel).
        """
        if action_type == 'read':
            cc_action = ActionType.READ
        elif action_type == 'write':
            cc_action = ActionType.WRITE
        else:
            raise ValueError("Tipe aksi tidak valid untuk CCM, harus 'read' atau
```

```
'write'")

        cc_row = Row(table_name=table_name, data={},object_id=table_name)


        return self.ccm.validate_object(cc_row, transaction_id, cc_action)

    def log_object(self, transaction_id: int, table_name: str):
        """
        Mencatat (log) objek/tabel yang diakses oleh transaksi ke CCM.
        """
        cc_row = Row(table_name=table_name, data={},object_id=table_name)


        self.ccm.log_object(cc_row, transaction_id)
```

2. Query Optimizer



Komponen Query Optimizer berfungsi menghasilkan rencana eksekusi yang lebih efisien dari sebuah query SQL. Query yang awalnya hanya berupa teks diubah menjadi *QueryTree*, lalu dianalisis serta ditransformasi untuk menurunkan biaya eksekusi. Proses optimasi ini menggabungkan pendekatan berbasis aturan dan pendekatan berbasis biaya untuk memastikan rencana yang dihasilkan memiliki estimasi I/O dan CPU paling rendah.

Query Optimizer terdiri dari beberapa komponen utama yang bekerja secara terurut. Komponen tersebut antara lain adalah sebagai berikut.

1. **OptimizationEngine**

   Menjadi titik masuk utama ketika Query Processor meminta proses parsing atau optimasi. Engine ini mengatur keseluruhan alur, mulai dari pembentukan token, parsing, validasi QueryTree, penerapan aturan heuristik, sampai memanggil GA jika diaktifkan.

2. **Tokenizer dan Parser**

   Tokenizer membagi SQL menjadi token, kemudian Parser menyusun token tersebut menjadi QueryTree sebagai representasi logis dari query. Struktur pohon inilah yang menjadi dasar proses optimasi.

3. **QueryTree**

   QueryTree memuat struktur operasi seperti RELATION, FILTER, JOIN, dan PROJECT. Seluruh transformasi dilakukan pada pohon ini, baik untuk pushdown maupun reorder JOIN.

4. **CostCalculator**

   Digunakan untuk menghitung estimasi biaya dari sebuah QueryTree. Komponen ini membaca statistik tabel dari Storage Manager, seperti jumlah baris dan jumlah blok, untuk memperkirakan biaya operasi seperti scan, filter, dan join.

5. **Query_check**

   Melakukan validasi sederhana untuk memastikan QueryTree hasil parsing tidak memiliki kesalahan struktur atau sintaks dasar.

Selanjutnya terdapat juga beberapa aturan (rule) yang digunakan dalam optimasi, beberapa aturan tersebut antara lain adalah sebagai berikut.

- normalisasi kondisi seleksi
- pemindahan kondisi WHERE ke ON saat JOIN
- pertukaran urutan JOIN (commutativity)

- perubahan struktur JOIN bertingkat (associativity)
- pushdown filter
- pushdown proyeksi

Aturan tersebut bisa dipakai langsung (heuristik awal) maupun sebagai bagian dari transformasi dalam GA.

Untuk query dengan banyak JOIN, ruang rencana eksekusi menjadi sangat besar. Oleh karena itu, digunakan Genetic Algorithm untuk mencari kombinasi aturan terbaik. GeneticOptimizer membangkitkan populasi awal, melakukan seleksi, crossover, dan mutasi.

Individual berisi parameter transformasi yang menentukan aturan apa yang diterapkan. Setiap individu menghasilkan QueryTree baru, lalu CostCalculator menghitung biayanya sebagai fitness. RuleParamsManager menyediakan fungsi untuk menganalisis QueryTree, membentuk parameter awal, dan memutasi parameter secara terkontrol. Dengan cara ini, Genetic Algorithm dapat menjelajahi banyak variasi rencana eksekusi tanpa harus mencoba semua kemungkinan.

Terakhir, berikut alur kerja Query Optimizer dan integrasinya dengan komponen lain.

- Parsing awal dan validasi

  Query diparse menjadi QueryTree, kemudian diperiksa oleh query_check untuk memastikan strukturnya valid sebelum proses optimasi dilakukan.

```
def parse_query(self, sql: str) -> ParsedQuery:
    self.original_sql = sql

    tokenizer = Tokenizer(sql)

    parser = Parser(tokenizer)
    query_tree = parser.parse()

    check_query(query_tree)
    result = ParsedQuery(query_tree, sql)
    return result
```

- Penerapan heuristik awal

Optimizer menerapkan aturan dasar seperti pushdown filter dan proyeksi untuk menyederhanakan QueryTree sejak awal.

```
# Rule 3: Projection elimination
from .rule.rule_3 import seleksi_proyeksi
query_tree = seleksi_proyeksi(query_tree)

# Rule 7: Filter pushdown over joins
from .rule.rule_7 import apply_pushdown
query_tree = apply_pushdown(query_tree)

# Rule 8: Projection push-down over joins
from .rule.rule_8 import push_projection_over_joins
query_tree = push_projection_over_joins(query_tree)

if use_genetic:
    from .genetic_optimizer import GeneticOptimizer

    ga = GeneticOptimizer(
        population_size=population_size,
        generations=generations,
        mutation_rate=mutation_rate,
        elitism=elitism or 2,
    )

    optimized_tree, history = ga.optimize(query_tree)
    self.optimized_tree = optimized_tree.query_tree
else:
    optimized_tree = query_tree
    self.optimized_tree = optimized_tree.query_tree

return optimized_tree
```

- Analisis QueryTree untuk kebutuhan GA

  Jika proses optimasi menggunakan Genetic Algorithm, QueryTree dianalisis untuk menentukan bagian yang dapat ditransformasi, terutama struktur JOIN.

- Proses Genetic Algorithm

  GeneticOptimizer membentuk populasi individu, kemudian menjalankan seleksi, crossover, dan mutasi. Setiap individu menghasilkan QueryTree baru dan biayanya dihitung menggunakan CostCalculator sebagai nilai fitness.

```
class Individual:
    def __init__(self, operation_params, base_query, lazy_eval=False, genealogy=None):
        self.operation_params = operation_params
        self.base_query = base_query
        self._query_cache = None
```

```
        self.fitness = None
        self.genealogy = genealogy or {}

        if not lazy_eval:
             self._query_cache = self._apply_transformations(base_query)

    @property
    def query(self):
        if self._query_cache is None:
            self._query_cache = self._apply_transformations(self.base_query)
        return self._query_cache


    def _inject_join_params_into_filter(self, filter_params, join_params):
        fp_complete = {k: copy.deepcopy(v) for k, v in filter_params.items()}

        join_target_ids = set()
        for target_list in join_params.values():
            join_target_ids.update(target_list)

        if not join_target_ids:
            return fp_complete

        for sig, order_spec in fp_complete.items():
            existing_ids = set()
            for item in order_spec:
                if isinstance(item, list):
                    existing_ids.update(item)
                else:
                    existing_ids.add(item)

            missing_ids = [x for x in sig if x in join_target_ids and x not in
existing_ids]

            for mid in missing_ids:
                order_spec.append(mid)

        return fp_complete

    def _apply_transformations(self, base_query):
        new_tree = base_query.query_tree.clone(deep=True, preserve_id=True)
        q = ParsedQuery(new_tree, base_query.query)

        fp_raw = self.operation_params.get('filter_params', {})
        jp = self.operation_params.get('join_params', {})
        jcp = self.operation_params.get('join_child_params', {})
        jap = self.operation_params.get('join_associativity_params', {})
        jmp = self.operation_params.get('join_method_params', {})

        fp_for_rule1 = self._inject_join_params_into_filter(fp_raw, jp)


        if fp_for_rule1:
            q, _ = rule_1_2.apply_rule1_rule2(q, fp_for_rule1)

        if jp:
            q, jp, fp_clean = rule_4.apply_merge(q, jp, fp_for_rule1)
            self.operation_params['join_params'] = jp
            self.operation_params['filter_params'] = fp_clean
```

```
        else:
            self.operation_params['filter_params'] = fp_for_rule1

        if jap:
            q = rule_6.apply_associativity(q, jap)

        if jcp:
            q, jcp = rule_5.apply_join_commutativity(q, jcp)
            self.operation_params['join_child_params'] = jcp

        if jmp:
            def apply_methods(node):
                if node.type == "JOIN" and node.id in jmp:
                    node.method = jmp[node.id]
                for child in node.childs:
                    apply_methods(child)
            apply_methods(q.query_tree)

        return q

class GeneticOptimizer:
    def __init__(self, population_size=50, generations=100, mutation_rate=0.1,
elitism=2):
        self.pop_size = population_size
        self.gens = generations
        self.mut_rate = mutation_rate
        self.elitism = elitism
        self.history = []

    def optimize(self, query):
        mgr = get_rule_params_manager()
        ops = mgr.get_registered_operations()

        base_analysis = {}
        for op in ops:
            base_analysis[op] = mgr.analyze_query(query, op)

        pop = []
        for _ in range(self.pop_size):
            params = {}
            for op, metadata in base_analysis.items():
                params[op] = {}
                for key, meta in metadata.items():
                    params[op][key] = mgr.generate_random_params(op, meta)
            pop.append(Individual(params, query))

        for g in range(self.gens):
            for ind in pop:
                if ind.fitness is None:
                    eng = OptimizationEngine()
                    ind.fitness = eng.get_cost(ind.query)

            pop.sort(key=lambda x: x.fitness)
            self.history.append({'gen': g, 'best': pop[0].fitness})

            next_pop = pop[:self.elitism]

            while len(next_pop) < self.pop_size:
                p1, p2 = random.sample(pop[:10], 2)
```

```
                c1, c2 = self._crossover(p1, p2, query)

                if random.random() < self.mut_rate: c1 = self._mutate(c1)
                if random.random() < self.mut_rate: c2 = self._mutate(c2)

                next_pop.extend([c1, c2])
            pop = next_pop[:self.pop_size]

        return pop[0].query, self.history

    def _crossover(self, p1, p2, base_query):
        c1_params = {}
        c2_params = {}
        c1_genealogy = {}
        c2_genealogy = {}

        coupled_ops = {'filter_params', 'join_params'}

        all_ops = set(p1.operation_params.keys()) | set(p2.operation_params.keys())

        inherit_group_from_p1 = random.choice([True, False])

        source_label_1 = "Parent A" if inherit_group_from_p1 else "Parent B"
        source_label_2 = "Parent B" if inherit_group_from_p1 else "Parent A"


        for op in coupled_ops:
            if op in all_ops:
                c1_params[op] = {}
                c2_params[op] = {}

                p1_data = p1.operation_params.get(op, {})
                p2_data = p2.operation_params.get(op, {})

                if inherit_group_from_p1:
                    c1_params[op] = {k: v for k,v in p1_data.items()}
                    c2_params[op] = {k: v for k,v in p2_data.items()}
                else:
                    c1_params[op] = {k: v for k,v in p2_data.items()}
                    c2_params[op] = {k: v for k,v in p1_data.items()}

                c1_genealogy[op] = f"[COUPLED] All from {source_label_1}"
                c2_genealogy[op] = f"[COUPLED] All from {source_label_2}"

        independent_ops = all_ops - coupled_ops

        for op in independent_ops:
            c1_params[op] = {}
            c2_params[op] = {}
            c1_genealogy[op] = {}
            c2_genealogy[op] = {}

            p1_data = p1.operation_params.get(op, {})
            p2_data = p2.operation_params.get(op, {})

            all_keys = set(p1_data.keys()) | set(p2_data.keys())

            for k in all_keys:
                val1 = p1_data.get(k)
```

```
                val2 = p2_data.get(k)

                src1 = "Parent A"
                src2 = "Parent B"

                if val1 is None:
                    val1 = val2
                    src1 = "Parent B"
                if val2 is None:
                    val2 = val1
                    src2 = "Parent A"

                if random.random() < 0.5:
                    c1_params[op][k] = val1
                    c1_genealogy[op][k] = src1

                    c2_params[op][k] = val2
                    c2_genealogy[op][k] = src2
                else:
                    c1_params[op][k] = val2
                    c1_genealogy[op][k] = src2
                    c2_params[op][k] = val1
                    c2_genealogy[op][k] = src1

        return (Individual(c1_params, base_query, lazy_eval=True,
genealogy=c1_genealogy),
                Individual(c2_params, base_query, lazy_eval=True,
genealogy=c2_genealogy))

    def _mutate(self, ind):
        new_params = {op: {k: v for k,v in data.items()} for op, data in
ind.operation_params.items()}
        new_genealogy = ind.genealogy.copy()

        mgr = get_rule_params_manager()

        if not new_params: return ind

        op = random.choice(list(new_params.keys()))
        data = new_params[op]

        if data:
            key = random.choice(list(data.keys()))
            data[key] = mgr.mutate_params(op, data[key])

            if isinstance(new_genealogy.get(op), dict):
                new_genealogy[op][key] = "MUTATED"
            else:
                current_info = new_genealogy.get(op, "")
                new_genealogy[op] = f"{current_info} + MUTATED ({key})"

        return Individual(new_params, ind.base_query, lazy_eval=True,
genealogy=new_genealogy)
```

- Pemilihan rencana terbaik

Setelah beberapa generasi, individu dengan biaya terendah dipilih. QueryTree milik individu tersebut digunakan sebagai hasil akhir optimasi.

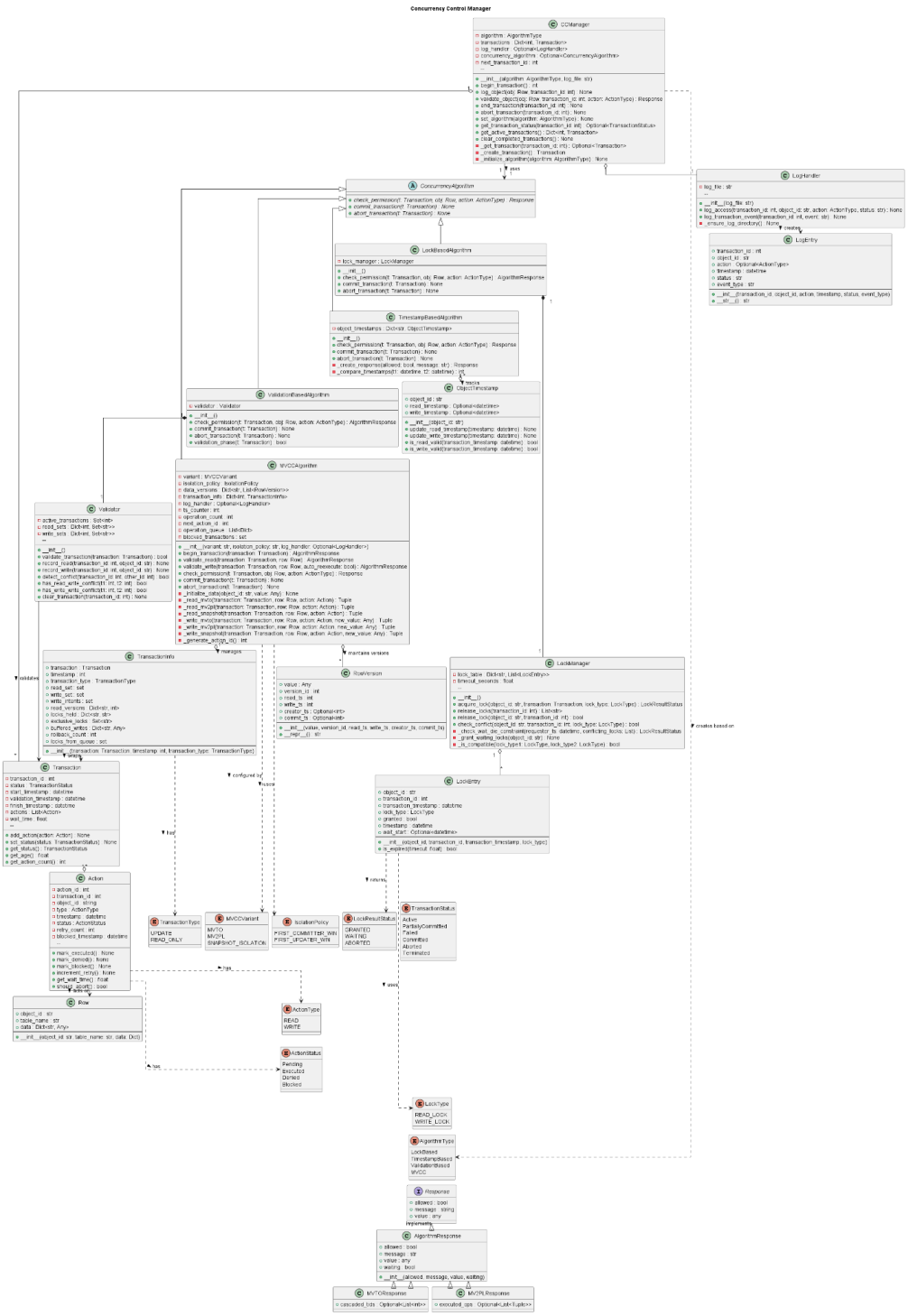
- Integrasi dengan Query Processor

  QueryTree hasil optimasi dikembalikan ke Query Processor untuk diterjemahkan menjadi rencana eksekusi fisik sesuai kebutuhan sistem.

- Ketergantungan pada Storage Manager

  CostCalculator menggunakan metadata dari Storage Manager, seperti jumlah blok dan jumlah record, untuk menghitung biaya secara akurat baik dalam heuristik maupun GA.

3. Concurrency Control Manager



Gambar x. Class Diagram Concurrency Control Manager

Rancangan di atas merupakan arsitektur Concurrency Control Manager (CCM) yang berfungsi untuk mengatur eksekusi transaksi agar tetap konsisten dan bebas dari konflik ketika beberapa transaksi berjalan secara paralel. Sistem ini dibangun secara modular dan extensible, terdiri dari komponen umum dan komponen khusus untuk setiap algoritma kontrol konkurensi (Lock-Based, Timestamp-Based, Validation-Based, dan MVCC).

1. Komponen Umum

Komponen ini bersifat independen dari algoritma, dan berfungsi sebagai pondasi utama sistem:

1.1. CCManager

Merupakan komponen atau antarmuka utama yang menyediakan abstraksi untuk mengatur siklus hidup transaksi mulai dari inisialisasi, validasi tindakan, hingga komit dan abort dan juga konkurensi transaksi. CCManager didesain menggunakan *Strategy Pattern* agar setiap algoritma yang diimplementasikan dapat diubah secara dinamis. Berikut merupakan komponen utama dari CCM:

- Algorithm Selection, CCM mendukung empat algoritma utama, yaitu Lock-Based, Timestamp-Based, Validation-Based, dan juga MVCC (Multiversion Concurrency Control).
- Transaction Registry, CCM menyimpan semua transaksi aktif dalam dictionary “transactions: Dict[int, Transaction]”.
- Algorithm Delegation, CCM mendelegasikan validasi operasi ke algoritma yang sedang aktif.
- Transaction ID Management, CCM mengelola ID unik setiap transaksi dengan counter auto-increment.

Adapun lifecycle state dari transaksi yang dipegang, seperti berikut.

```
Active → PartiallyCommitted → Committed → Terminated
   ↓                  ↓
Failed → Aborted → Terminated
```

Berikut merupakan cuplikan kode implementasi inisiasi dari CCM, dengan _initalize_algorithm merupakan fungsi initialize yang akan memanggil instance dari algoritma yang sedang aktif.

```
def __init__(self, algorithm: AlgorithmType, log_file: str = "cc_log.txt"):
    self.algorithm: AlgorithmType = algorithm
    self.transactions: Dict[int, Transaction] = {}
    self.log_handler: Optional[LogHandler] = LogHandler(log_file)
    self.concurrency_algorithm: Optional[ConcurrencyAlgorithm] = None
    self.next_transaction_id: int = 1
    self._initialize_algorithm(algorithm)
```

Desain dari kode ini memungkinkan loose coupling antara implementasi algoritma dan juga algoritma dapat berubah secara dinamis.

Selain dari itu terdapat juga beberapa fungsi inti dari CCM, seperti begin_transaction yang berguna untuk memulai suatu transaksi dan melakukan assignment id pada transaksi tersebut. Kemudian terdapat fungsi validasi operasi yang akan mengecek status dari transaksi dan menentukan status selanjutnya dengan mendelegasikan tugasnya ke algoritma yang sedang aktif, berikut merupakan implementasinya.

```
def validate_object(self, obj: Row, transaction_id: int, action: ActionType) ->
Response:
    transaction = self._get_transaction(transaction_id)

    # Validasi status transaksi
    if transaction.get_status() not in [TransactionStatus.Active,
TransactionStatus.PartiallyCommitted]:
        return Response(allowed=False, message="Transaction not in active state")

    # Delegasi ke algoritma aktif
    response = self.concurrency_algorithm.check_permission(transaction, obj, action)

    # Logging hasil
    status = "ALLOWED" if response.allowed else "DENIED"
    self.log_handler.log_access(transaction_id, obj.object_id, action, status)

    return response
```

Fitur terakhir dari CCM adalah end_transaction yang berguna untuk mengakhiri suatu transaksi dengan fase pertama status akan masuk ke status PartiallyCommitted, kedua algoritma akan melakukan commit atau abort jika terjadi error. Setelah selesai maka transaksi akan ditandai sebagai Terminated.

```
def end_transaction(self, transaction_id: int) -> None:
```

```
    transaction = self._get_transaction(transaction_id)
    current_status = transaction.get_status()

    if current_status == TransactionStatus.Active:
        # Phase 1: Prepare (PartiallyCommitted)
        transaction.set_status(TransactionStatus.PartiallyCommitted)
        self.log_handler.log_transaction_event(transaction_id, "PARTIALLY_COMMITTED")

        # Phase 2: Commit
        try:
            self.concurrency_algorithm.commit_transaction(transaction)
            transaction.set_status(TransactionStatus.Committed)
            self.log_handler.log_transaction_event(transaction_id, "COMMITTED")
        except Exception as e:
            # Rollback
            transaction.set_status(TransactionStatus.Failed)
            self.concurrency_algorithm.abort_transaction(transaction)
            transaction.set_status(TransactionStatus.Aborted)
            raise e

    # Mark as terminated
    transaction.set_status(TransactionStatus.Terminated)
```

1.2. Transaction dan Action

Komponen Transaction mewakili transaksi individual dan komponen Action mewakili operasi (seperti READ/WRITE) yang dilakukan oleh transaksi tersebut. Setiap transaksi mempunyai ID transaksi, status transaksi (*active*, *partially committed*, *aborted*, dan *terminated*), *timestamp* (waktu mulai, waktu validasi, dan waktu selesai), daftar aksi, dan waktu tunggu. Sedangkan itu, setiap aksi mempunyai ID aksi, ID transaksi, ID objek, tipe aksi (READ atau WRITE), status aksi (pending, executed, denied, blocked), timestamp ketika berada di status blocked, dan jumlah *retry*. Terdapat maksimal jumlah *retry* yang dilakukan oleh sebuah aksi, yaitu tiga kali. Hal ini untuk menentukan kapan sebuah aksi harus di-*abort*. Fungsi yang berada pada kedua komponen merupakan fungsi-fungsi dasar seperti get dan set. Fungsi lainnya adalah sebagai berikut.

```
def should_abort(self) -> bool:
        """Check if action should be aborted"""
        return self.retry_count >= MAX_RETRY
```

1.3. LogHandler dan LogEntry

Berfungsi sebagai komponen pengawas yang menyimpan catatan aktivitas transaksi seperti pembacaan, penulisan, dan peristiwa commit/abort. Berbeda

dengan Log yang dimiliki pada modul Failure Recovery yang berfungsi untuk melakukan recovery ketika terdapat transaksi yang batal, LogHandler ini digunakan untuk debugging, recovery, auditing, dan juga performance analysis pada internal CCM. LogHandler ini juga tidak menjadi dependency pada algoritma manapun sehingga bersifat independen. Setiap entry log akan memiliki timestamp dengan presisi mikrodetik. Berikut merupakan cuplikan implementasi dari LogHandler.

```
def log_transaction_event(self, transaction_id: int, event: str) -> None:
    entry = LogEntry(
        timestamp=datetime.now(),
        transaction_id=transaction_id,
        event_type=event,
        message=f"Transaction {transaction_id}: {event}"
        )
    self.entries.append(entry)
    self._write_to_file(entry)

def log_access(self, transaction_id: int, object_id: str,
               action: ActionType, status: str) -> None:
    entry = LogEntry(
        timestamp=datetime.now(),
        transaction_id=transaction_id,
        event_type=action.value,
        object_id=object_id,
        status=status,
        message=f"T{transaction_id} {action.value} {object_id}: {status}"
    )
    self.entries.append(entry)
    self._write_to_file(entry)
```

LogHandler akan menghasilkan hasil format log seperti berikut.

```
[2025-11-13 16:55:29.724069] T1 | BEGIN_TRANSACTION | Object:  |
Action: N/A | Status:
[2025-11-13 16:55:29.725080] T1 | ACCESS | Object: account_1 |
Action: READ | Status: LOGGED
[2025-11-13 16:55:29.727078] T1 | ACCESS | Object: account_1 |
Action: READ | Status: ALLOWED
[2025-11-13 16:55:29.728078] T1 | ACCESS | Object: account_1 |
Action: WRITE | Status: ALLOWED
```

1.4. Row

Representasi objek data yang diakses oleh transaksi. Menyimpan nilai dan informasi tabel terkait objek tersebut.

2. Komponen Algoritma Kontrol Konkurensi

Semua algoritma diturunkan dari abstract class ConcurrencyAlgorithm, yang menyediakan antarmuka dasar seperti check_permission() akan menentukan apakah aksi boleh dieksekusi, commit_transaction() dan abort_transaction() akan menangani akhir transaksi.

2.1. Abstract Algorithm Class

Untuk memastikan keselarasan dari semua algoritma yang akan diimplementasikan maka dibuat kelas abstract Concurrency Algorithm yang menyediakan abstract method yang harus diimplementasikan oleh setiap algoritma. Algoritma tersebut terdiri dari check_permission untuk mengecek apakah suatu transaksi dapat berjalan, commit_transaction untuk melakukan commit pada transaksi, dan juga abort_transaction jika terdapat error.

```
class ConcurrencyAlgorithm(ABC):

    @abstractmethod
    def check_permission(self, t: Transaction, obj: Row,
                         action: ActionType) -> Response:
        """Check if transaction has permission to perform action on object"""
        pass

    @abstractmethod
    def commit_transaction(self, t: Transaction) -> None:
        """Commit a transaction"""
        pass

    @abstractmethod
    def abort_transaction(self, t: Transaction) -> None:
        """Abort a transaction"""
        pass
```

2.2. Lock-Based Algorithm

Menggunakan mekanisme penguncian untuk mengatur akses data, dengan LockManager mengelola tabel kunci (lock_table). Sedangkan LockEntry menyimpan informasi setiap kunci yang sedang dipegang transaksi.

Skema wait-die digunakan pada algoritma lock-based ini untuk menghindari terjadinya deadlock. Setiap transaksi yang dimulai nantinya akan diberikan timestamp, sehingga di antara 2 transaksi, dapat diketahui siapa yang lebih tua

dan siapa yang lebih muda. Saat transaksi yang lebih tua meminta akses kunci row yang dipegang transaksi lebih muda, maka dia akan menunggu sampai kunci dilepaskan (Wait), sedangkan jika transaksi yang lebih muda meminta akses kunci row yang dipegang transaksi lebih tua, maka transaksi ini akan langsung di-abort (Die). Dengan cara inilah deadlock dapat dihindari.

```
conflicting_locks : List[LockEntry] = []
     for lock in lock_list:
         if lock.transaction_id != transaction_id and lock.granted == True:
             if self._is_compatible(lock.lock_type, lock_type) == False:
                 conflicting_locks.append(lock)
     if existing_locks:
         if existing_locks.granted:
             if existing_locks.lock_type == lock_type or
                 existing_locks.lock_type == LockType.WRITE_LOCK:
                 return LockResultStatus.GRANTED

             # Upgrade Case: Read -> Write
             if existing_locks.lock_type == LockType.READ_LOCK and lock_type
                  == LockType.WRITE_LOCK:
                 if conflicting_locks:
                     # Wait-die logic
                     decision = self._check_wait_die_constraint
                        (transaction_ts, conflicting_locks)
                     if decision == LockResultStatus.ABORTED:
                         return LockResultStatus.ABORTED

                     existing_locks.granted = False
                     existing_locks.wait_start = datetime.now()
                     return LockResultStatus.WAITING


 def _check_wait_die_constraint(self, requester_ts: datetime,
      conflicting_locks: List[LockEntry]) -> LockResultStatus:

     for holder in conflicting_locks:
         holder_ts = holder.transaction_timestamp

         # Requester lebih Muda
         if requester_ts > holder_ts:
             return LockResultStatus.ABORTED
         else:
             return LockResultStatus.WAITING
```

2.3. Timestamp-Based Algorithm

Algoritma Timestamp-Based menggunakan timestamp untuk menjaga urutan serial transaksi dengan ObjectTimestamp menyimpan waktu *read* dan *write* tiap objek data. Pada implementasi dan *approach* ini, setiap transaksi diberi timestamp

unik saat dimulai, dan sistem memastikan bahwa eksekusi transaksi menghasilkan jadwal yang setara dengan eksekusi serial berdasarkan urutan timestamp.

1. Struktur Data ObjectTimestamp

   Kelas ObjectTimestamp digunakan untuk melacak timestamp baca (read timestamp) dan timestamp tulis (write timestamp) untuk setiap objek data:

```python
class ObjectTimestamp:
      """Tracks read and write timestamps for an object"""

      def __init__(self, object_id: str):
          self.object_id: str = object_id
          self.read_timestamp: Optional[datetime] = None
          self.write_timestamp: Optional[datetime] = None

      def update_read_timestamp(self, timestamp: datetime) -> None:
          """Update read timestamp to max of current and new timestamp"""
          if self.read_timestamp is None or timestamp > self.read_timestamp:
              self.read_timestamp = timestamp

      def update_write_timestamp(self, timestamp: datetime) -> None:
          """Update write timestamp"""
          self.write_timestamp = timestamp
```

Penjelasan:
Setiap objek data memiliki dua atribut timestamp:

- read_timestamp (R-TS): Timestamp terbesar dari semua transaksi yang berhasil membaca objek tersebut
- write_timestamp (W-TS): Timestamp dari transaksi terakhir yang berhasil menulis objek tersebut

2. Validasi *Read and Write Operations*

   Kelas ObjectTimestamp menyediakan dua metode untuk memvalidasi apakah operasi baca atau tulis diizinkan:

```python
def is_read_valid(self, transaction_timestamp: datetime) -> bool:
      """Check if read is valid for given transaction timestamp
      Read is valid if:
      - TS(T) >= W-timestamp(X)
      (Transaction can read if it's newer than the last write)
      """
      if self.write_timestamp is None:
          return True
      return transaction_timestamp >= self.write_timestamp

  def is_write_valid(self, transaction_timestamp: datetime) -> bool:
      """Check if write is valid for given transaction timestamp

      Write is valid if:
      - TS(T) >= R-timestamp(X) (newer than last read)
      - TS(T) >= W-timestamp(X) (newer than last write)
      """
```

```
        # Check against read timestamp
        if self.read_timestamp is not None and transaction_timestamp <
self.read_timestamp:
            return False

        # Check against write timestamp
        if self.write_timestamp is not None and transaction_timestamp <
self.write_timestamp:
            return False

        return True
```

Penjelasan:
Aturan validasi berdasarkan protokol timestamp ordering:

- Operasi READ: Valid jika TS(T) ≥ W-TS(X), artinya transaksi hanya dapat membaca data jika timestamp-nya lebih baru atau sama dengan timestamp penulisan terakhir
- Operasi WRITE: Valid jika TS(T) ≥ R-TS(X) dan TS(T) ≥ W-TS(X), artinya transaksi hanya dapat menulis jika timestamp-nya lebih baru dari timestamp pembacaan dan penulisan terakhir

3. Implementasi TimestampBasedAlgorithm

   Kelas TimestampBasedAlgorithm mengimplementasikan antarmuka ConcurrencyAlgorithm dan mengelola kontrol concurrency berbasis timestamp:

```
class TimestampBasedAlgorithm(ConcurrencyAlgorithm):
        """Timestamp-based concurrency control algorithm implementation"""

        def __init__(self):
            self.object_timestamps: Dict[str, ObjectTimestamp] = {}
```

Penjelasan:
Atribut object_timestamps adalah dictionary yang menyimpan ObjectTimestamp untuk setiap object data yang diakses dengan object_id.

4. Method check_permission

   Method check_permission merupakan bagian dari *timestamp-based algorithm* yang memvalidasi dan memproses permintaan akses:

```
def check_permission(self, t: Transaction, obj: Row,
                     action: ActionType) -> Response:
        object_id = obj.object_id
        transaction_timestamp = t.start_timestamp

        # Get or create object timestamp
        if object_id not in self.object_timestamps:
            self.object_timestamps[object_id] = ObjectTimestamp(object_id)

        obj_ts = self.object_timestamps[object_id]
```

```
      # Check permission based on action type
      if action == ActionType.READ:
          if obj_ts.is_read_valid(transaction_timestamp):
              # Update read timestamp
              obj_ts.update_read_timestamp(transaction_timestamp)
              return self._create_response(
                  allowed=True,
                  message=f"Read allowed for transaction {t.transaction_id} on
object {object_id}"
              )
          else:
              return self._create_response(
                  allowed=False,
                  message=f"Read denied for transaction {t.transaction_id} on
object {object_id}: "
                          f"Transaction timestamp {transaction_timestamp} <
Write timestamp {obj_ts.write_timestamp}"
              )

      elif action == ActionType.WRITE:
          if obj_ts.is_write_valid(transaction_timestamp):
              # Update write timestamp
              obj_ts.update_write_timestamp(transaction_timestamp)
              return self._create_response(
                  allowed=True,
                  message=f"Write allowed for transaction {t.transaction_id} on
object {object_id}"
              )
          else:
              # Determine reason for rejection
              if obj_ts.read_timestamp and transaction_timestamp <
obj_ts.read_timestamp:
                  reason = f"Transaction timestamp {transaction_timestamp} <
Read timestamp {obj_ts.read_timestamp}"
              else:
                  reason = f"Transaction timestamp {transaction_timestamp} <
Write timestamp {obj_ts.write_timestamp}"

              return self._create_response(
                  allowed=False,
                  message=f"Write denied for transaction {t.transaction_id} on
object {object_id}: {reason}"
              )
```

Penjelasan:

1. Mengambil object_id dari objek yang diakses dan start_timestamp dari transaksi
2. Membuat atau mengambil objek ObjectTimestamp untuk objek data tersebut
3. Untuk operasi READ:
    a. Jika valid (TS(T) ≥ W-TS(X)), perbarui R-TS(X) dan izinkan operasi
    b. Jika tidak valid, tolak operasi karena data sudah ditulis oleh transaksi yang lebih baru
4. Untuk operasi WRITE:
    a. Jika valid (TS(T) ≥ R-TS(X) dan TS(T) ≥ W-TS(X)), perbarui W-TS(X) dan izinkan operasi
    b. Jika tidak valid, tolak operasi dengan pesan yang menjelaskan alasan penolakan

5. Method commit_transaction dan abort_transaction

```
def commit_transaction(self, t: Transaction) -> None:
      """Commit transaction

      For timestamp-based algorithm, no special action needed on commit
      as timestamps are already updated during operation validation
      """
      pass

  def abort_transaction(self, t: Transaction) -> None:
      """Abort transaction

      For timestamp-based algorithm, we don't rollback timestamps
      as the transaction never completed. The timestamps of objects
      remain as they were set by successfully committed transactions.
      """
      pass
```

Penjelasan:

- commit_transaction: Tidak memerlukan aksi khusus karena timestamp sudah diperbarui saat validasi operasi
- abort_transaction: Tidak melakukan rollback timestamp karena timestamp yang tercatat merepresentasikan operasi dari transaksi yang berhasil dieksekusi

6. Method Helper

```
def _get_read_timestamp(self, object_id: str) -> Optional[datetime]:
      """Get read timestamp for an object"""
      if object_id in self.object_timestamps:
          return self.object_timestamps[object_id].read_timestamp
      return None

  def _get_write_timestamp(self, object_id: str) -> Optional[datetime]:
      """Get write timestamp for an object"""
      if object_id in self.object_timestamps:
          return self.object_timestamps[object_id].write_timestamp
      return None

  def _update_timestamps(self, object_id: str, t: Transaction,
                         action: ActionType) -> None:
      """Update timestamps after successful operation"""
      if object_id not in self.object_timestamps:
          self.object_timestamps[object_id] = ObjectTimestamp(object_id)

      obj_ts = self.object_timestamps[object_id]

      if action == ActionType.READ:
          obj_ts.update_read_timestamp(t.start_timestamp)
      elif action == ActionType.WRITE:
          obj_ts.update_write_timestamp(t.start_timestamp
```

Penjelasan:
Methods ini menyediakan akses terstruktur ke timestamp objek dan memudahkan pembaruan timestamp setelah operasi berhasil.

2.4. Validation-Based Algorithm

Merupakan salah satu dari teknik kontrol konkurensi yang menganggap konflik merupakan hal yang jarang terjadi sehingga semua aksi/operasi diperbolehkan selama waktu eksekusi (*optimistic concurrency control*). Pengecekan konflik hanya dilakukan ketika sebuah transaksi melakukan *commit*. Implementasi algoritma dilakukan dengan penyimpanan transaksi yang sedang aktif, *read set*, dan *write set* untuk transaksi tersebut. Read set dan write set didapatkan dengan menggunakan dua fungsi, yaitu record_read() dan record_write(). Berikut adalah contoh dari penyimpanan.

```
# T1 membaca A dan B serta menulis C
active_transactions = {1}
read_sets = {1:{'A','B'}}
write_sets = {1:{'C'}}
```

Dengan menggunakan contoh T1 dan T2, konflik dapat terjadi ketika T1 membaca objek yang T2 tulis, T1 menulis objek yang T2 tulis, dan sebaliknya. Berikut adalah fungsi-fungsi yang menjadi pusat dari algoritma ini.

```
class Validator:
    """Validates transactions for optimistic concurrency control"""

    def __init__(self):
        self.active_transactions: Set[int] = set()
        self.read_sets: Dict[int, Set[str]] = {}
        self.write_sets: Dict[int, Set[str]] = {}

    def validate_transaction(self, transaction: Transaction) -> bool:
        """Validate a transaction before commit"""
        transaction_id = transaction.transaction_id
        # Get all other active transactions (excluding self)
        other_transactions = self.active_transactions - {transaction_id}
        # Check for conflicts with each other active transaction
        for other_id in other_transactions:
            if self.detect_conflict(transaction_id, other_id):
                return False
        return True

   def detect_conflict(self, transaction_id: int, other_id: int) -> bool:
        """Detect conflict between two transactions"""
        if self.has_read_write_conflict(transaction_id, other_id):
            return True
        if self.has_read_write_conflict(other_id, transaction_id):
            return True
        if self.has_write_write_conflict(transaction_id, other_id):
            return True
        return False
```

```python
class ValidationBasedAlgorithm(ConcurrencyAlgorithm):
    """Validation-based (Optimistic) concurrency control algorithm implementation"""

    def __init__(self):
        self.validator: Validator = Validator()
   def validation_phase(self, t: Transaction) -> bool:
        """Perform validation phase before commit"""
        return self.validator.validate_transaction(t)
   def check_permission(self, t: Transaction, obj: Row,
                        action: ActionType) -> AlgorithmResponse:
        """Check permission (always allowed in read/write phase)"""
        transaction_id = t.transaction_id
        object_id = obj.object_id

        # Conflicts are detected only during validation phase
        if action == ActionType.READ:
            self.validator.record_read(transaction_id, object_id)
            return AlgorithmResponse(
                allowed=True,
                message=f"T{transaction_id} reads {object_id} - allowed (optimistic)"
            )
        elif action == ActionType.WRITE:
            self.validator.record_read(transaction_id, object_id)
            self.validator.record_write(transaction_id, object_id)
            return AlgorithmResponse(
                allowed=True,
                message=f"T{transaction_id} writes {object_id} - allowed (optimistic)"
            )
        else:
            return AlgorithmResponse(
                allowed=False,
                message=f"Unknown action type: {action}"
            )

    def commit_transaction(self, t: Transaction) -> None:
        """Commit transaction after validation"""
        from datetime import datetime

        t.validation_timestamp = datetime.now()

        # Perform validation
        if self.validation_phase(t):
            # commit n clear the transaction data if success
            t.set_status(TransactionStatus.Committed)
            self.validator.clear_transaction(t.transaction_id)
        else:
            # t.set_status(TransactionStatus.Aborted)
            self.abort_transaction(t)
```

2.5. MVCC (Multiversion Concurrency Control)

Multi-Version Concurrency Control (MVCC) merupakan teknik kontrol konkurensi yang memungkinkan berbagai versi dari suatu item data berada secara bersamaan dalam basis data. Pendekatan ini memungkinkan operasi pembacaan mengakses versi data yang lebih lama, sementara operasi penulisan menghasilkan

versi baru, sehingga tingkat konkurensi dalam sistem dapat ditingkatkan. Berikut merupakan keuntungan algoritma MVCC:

1. Konkurensi Lebih Tinggi

   Operasi pembacaan tidak memblokir operasi penulisan, begitu juga sebaliknya sehingga aktivitas transaksi dapat berjalan lebih paralel.

2. Snapshot Isolation

   Setiap transaksi dapat beroperasi pada snapshot basis data yang konsisten, memastikan bahwa data yang dibaca tidak terpengaruh oleh perubahan yang dilakukan transaksi lain yang belum selesai.

Berikut merupakan struktur data utama untuk implementasi MVCC.

```
class RowVersion:
    def __init__(self, value: Any, version_id: int, read_ts: int, write_ts: int,
                 creator_ts: Optional[int] = None, commit_ts: Optional[int] = None):
        self.value: Any = value
        self.version_id: int = version_id
        self.read_ts: int = read_ts
        self.write_ts: int = write_ts
        self.creator_ts: Optional[int] = creator_ts
        self.commit_ts: Optional[int] = commit_ts
```

Keterangan:

- `RowVersion` menyimpan satu versi dari data item.
- `read_ts`: Timestamp transaksi terbesar yang membaca versi ini.
- `write_ts`: Timestamp transaksi yang membuat versi ini.
- `creator_ts` dan `commit_ts`: Digunakan untuk Snapshot Isolation.

```
class TransactionInfo:
    def __init__(self, transaction: Transaction, timestamp: int,
                 transaction_type: TransactionType = TransactionType.UPDATE):
        self.transaction: Transaction = transaction
        self.timestamp: int = timestamp
        self.read_set: set = set()
        self.write_set: set = set()
        self.read_versions: Dict[str, int] = {}
        self.locks_held: Dict[str, str] = {}
        self.exclusive_locks: Set[str] = set()
        self.buffered_writes: Dict[str, Any] = {}
```

Keterangan:

- `TransactionInfo` menyimpan metadata setiap transaksi.
- `read_set` dan `write_set`: digunakan untuk melacak operasi pembacaan dan penulisan yang dilakukan transaksi, sebagai dasar dalam pendeteksian konflik.
- `read_versions`: pemetaan antara `object_id` dan `write timestamp` dari versi data yang dibaca oleh transaksi yang digunakan untuk mencegah terjadinya cascading rollback.
- `buffered_writes`: menyimpan operasi penulisan secara sementara (buffer) hingga transaksi melakukan commit, khususnya pada mekanisme Snapshot Isolation.

Terdapat beberapa variasi algoritma untuk Multi-Version Concurrency Control, di antaranya Multi-Version Timestamp Ordering (MVTO), Multi-Version Two Phase Locking (MV2PL), dan Snapshot Isolation dengan variasi first-comitter win dan first-updater win. Berikut merupakan implementasi untuk algoritma-algoritma tersebut.

a. Multi-Version Timestamp Ordering (MVTO)

Multiversion Timestamp Ordering (MVTO) merupakan salah satu varian dari MVCC yang menggunakan timestamp untuk menentukan urutan serialisasi transaksi. Setiap transaksi diberikan timestamp unik pada saat transaksi dimulai (begin). Prinsip dasar untuk algoritma MVTO adalah sebagai berikut.

Aturan Pembacaan (Read Rule)

Transaksi T_i membaca versi objek data Q_k terbaru di mana nilai Write Timestamp dari versi tersebut memenuhi:

$$W - TS(Q_k) \leq TS(T_i)$$

Dengan demikian, transaksi hanya dapat membaca versi yang dibuat sebelum atau pada saat transaksi tersebut dimulai.

Aturan Penulisan (Write Rule)

Saat transaksi T_i ingin menulis objek data Q:

1. Jika $TS(T_i) < R - TS(Q_k)$ maka transaksi dibatalkan (abort) karena terdapat transaksi yang lebih baru telah membaca versi lama dari objek tersebut.
2. Jika $TS(T_i) = W - TS(Q_k)$, maka versi yang ada dapat ditimpa (overwrite) oleh transaksi tersebut.
3. Jika tidak memenuhi kedua kondisi di atas, dibuat versi baru dari objek data Q dengan: $W - TS = TS(T_i)$

Cascading Rollback

Apabila transaksi T_i dibatalkan, maka seluruh transaksi lain yang telah membaca versi data yang ditulis oleh T_i juga harus dibatalkan. Hal ini dilakukan untuk menjaga konsistensi versi data dalam basis data.

Berikut merupakan strategi pemgembangan yang dilakukan untuk algoritma MVTO.

1. Begin Transaction - Inisialisasi dengan Timestamp

```
def begin_transaction(self, transaction: Transaction) -> AlgorithmResponse:
    if self.variant == MVCCVariant.MVTO:
        timestamp = transaction.transaction_id
    else:
        timestamp = self.ts_counter

    trans_info = TransactionInfo(
        transaction=transaction,
        timestamp=timestamp,
        transaction_type=TransactionType.UPDATE
    )
    self.transaction_info[transaction.transaction_id] = trans_info
    transaction.set_status(TransactionStatus.Active)
```

Penjelasan:

- MVTO menggunakan transaction ID sebagai timestamp yang bersifat statis (static timestamp), sehingga urutan waktu setiap transaksi sudah ditentukan sejak transaksi tersebut dimulai.
- Struktur TransactionInfo digunakan untuk melacak status dan metadata transaksi selama eksekusi, termasuk informasi yang berkaitan dengan operasi pembacaan maupun penulisan.
- Status transaksi diinisialisasi sebagai Active pada saat transaksi mulai berjalan, sebagai penanda bahwa transaksi tersebut masih dalam proses dan belum mencapai tahap commit maupun abort.

2. Read Operation - Pilih Versi yang Tepat

```
def validate_read_mvto(self, transaction: Transaction, row: Row) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0)
        ]

    versions = self.data_versions[object_id]

    suitable_version = None
    for version in sorted(versions, key=lambda v: v.write_ts, reverse=True):
        if version.write_ts <= trans_info.timestamp:
            suitable_version = version
            break

    if suitable_version:
        if trans_info.timestamp > suitable_version.read_ts:
            suitable_version.read_ts = trans_info.timestamp

        trans_info.read_set.add(object_id)
        trans_info.read_versions[object_id] = suitable_version.write_ts

        return AlgorithmResponse(
            allowed=True,
            message=f"T{transaction.transaction_id} reads {object_id}{suitable_version.version_id}, "
                    f"R-TS updated to {suitable_version.read_ts}",
            value=suitable_version.value
        )
```

Penjelasan:

- Inisialisasi: jika suatu objek belum pernah diakses, buat versi awal (V_0) dengan write_ts = 0 dan read_ts = 0.
- Seleksi versi: pilih versi terbaru dari objek yang memenuhi W-TS $\leq$ TS(T_i).
- Perbarui R-TS: set read_ts versi menjadi max(read_ts, TS(T_i)). Dalam implementasi diperlakukan sebagai penggantian bila TS(T_i) lebih besar.
- Tracking untuk deteksi konflik/rollback:
  - Tambahkan object_id ke read_set transaksi.
  - Simpan read_versions[object_id] = W-TS(versi_dibaca) untuk keperluan deteksi cascading rollback.
- Keluaran: kembalikan respons yang memuat nilai versi yang dibaca dan pesan status (mis. pembaruan R-TS).

3. Write Operation - Conflict Detection & Versioning

```
def validate_write_mvto(self, transaction: Transaction, row: Row,
                        auto_reexecute: bool = True) -> MVTOResponse:
    action = Action(
        action_id=self.next_action_id,
        transaction_id=transaction.transaction_id,
```

```
            action_type=ActionType.WRITE,
            object_id=row.object_id,
            timestamp=datetime.now()
        )
        self.next_action_id += 1

        success, message = self._write_mvto(transaction, row, action)

        if not success:
            trans_info = self.transaction_info[transaction.transaction_id]
            abort_message = message

            is_first_operation = (len(trans_info.read_set) == 0 and
                                  len(trans_info.write_set) == 0)

            transactions_dict = {tid: info.transaction
                                 for tid, info in self.transaction_info.items()}
            new_ts, cascaded_tids = self.rollback_mvto_transaction(
                transaction, transactions_dict
            )

            if is_first_operation:
                success, message = self._write_mvto(transaction, row, action)
                response = MVTOResponse(allowed=success, message=message,
                                        cascaded_tids=cascaded_tids)
                response.abort_message = abort_message
                return response

            return MVTOResponse(allowed=False, message=abort_message,
                                cascaded_tids=cascaded_tids)

        return MVTOResponse(allowed=success, message=message, cascaded_tids=None)
```

Penjelasan:

- Deteksi Operasi Pertama
  Sistem memeriksa apakah read_set dan write_set pada transaksi masih kosong. Jika keduanya belum berisi apa pun, maka operasi yang sedang dilakukan dianggap sebagai operasi pertama dalam transaksi tersebut.
- Eksekusi Ulang Otomatis (Auto Re-execution)
  Apabila transaksi mengalami abort pada operasi pertama, sistem secara otomatis melakukan eksekusi ulang operasi tersebut dengan timestamp yang baru. Hal ini memastikan bahwa transaksi memiliki peluang untuk berhasil dieksekusi tanpa intervensi dari luar.
- Restart Manual
  Jika transaksi mengalami abort bukan pada operasi pertama (artinya sudah ada jejak pembacaan atau penulisan sebelumnya), maka sistem tidak melakukan pengulangan otomatis. Pada kondisi ini, caller wajib melakukan restart transaksi secara manual dari awal untuk menjaga konsistensi eksekusi.
- Pelacakan Cascading Rollback
  Sistem melacak seluruh transaksi yang terdampak oleh pembatalan transaksi lain (misalnya transaksi yang telah membaca versi yang ditulis oleh transaksi yang di-rollback). Semua transaksi yang terkena efek cascading akan dikumpulkan dan dikembalikan melalui objek respons, sehingga dapat diproses oleh lapisan yang lebih tinggi.

4. Rollback dan Cascading Abort

```
def _write_mvto(self, transaction: Transaction, row: Row,
                action: Action) -> Tuple[bool, str]:
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0)
        ]

    versions = self.data_versions[object_id]

    target_version = None
    for version in sorted(versions, key=lambda v: v.write_ts, reverse=True):
        if version.write_ts <= trans_info.timestamp:
            target_version = version
            break

    if not target_version:
        return False, f"No suitable version for write (TS={trans_info.timestamp})"

    if trans_info.timestamp < target_version.read_ts:
        rollback_info = f"(after rollback #{trans_info.rollback_count + 1})"
        message = (f"T{transaction.transaction_id} ABORTED: "
                   f"TS({trans_info.timestamp}) < R-TS({target_version.read_ts})")
        return False, message

    if trans_info.timestamp == target_version.write_ts:
        target_version.value = row.data.get('value', 0)
        message = f"T{transaction.transaction_id} overwrites {object_id}"
    else:
        new_version = RowVersion(
            value=row.data.get('value', 0),
            version_id=len(versions),
            read_ts=trans_info.timestamp,
            write_ts=trans_info.timestamp
        )
        self.data_versions[object_id].append(new_version)
        message = f"T{transaction.transaction_id} creates {object_id}{new_version.version_id}"

    trans_info.write_set.add(object_id)
    return True, message
```

Penjelasan:

- Pemberian Timestamp Baru
  Transaksi yang akan dieksekusi ulang diberikan timestamp baru yang bernilai lebih besar dibandingkan seluruh timestamp transaksi aktif yang ada, sehingga urutan serialisasi tetap terjaga.
- Pembersihan State
  Seluruh kondisi transaksi sebelumnya dihapus, termasuk read set, write set, locks, serta metadata lain yang berkaitan dengan eksekusi sebelum terjadi rollback. Dengan demikian, transaksi benar-benar kembali ke kondisi awal.
- Penghapusan Versi Data

Versi data yang telah dibuat oleh transaksi tersebut dihapus kembali dari sistem karena dianggap tidak valid akibat pembatalan transaksi.

- Deteksi Dampak Cascading
  Sistem menelusuri transaksi lain yang telah membaca versi data dari transaksi yang dibatalkan. Transaksi-transaksi tersebut dinyatakan terdampak dan tidak lagi konsisten.
- Rollback Rekursif
  Semua transaksi yang terpengaruh cascading akan di-rollback secara rekursif, kemudian penanganan rollback pada masing-masing transaksi dilakukan menggunakan prosedur yang sama hingga tidak ada lagi transaksi yang bergantung pada versi data yang telah dihapus.

b. Multi-Version Two Phase Locking (MV2PL)

Multi-Version Two-Phase Locking (MV2PL) merupakan teknik kontrol konkurensi yang mengombinasikan pendekatan multi-versioning dengan protokol two-phase locking (2PL). Pada mekanisme ini, transaksi bersifat read-only dapat membaca versi lama dari suatu item data tanpa memerlukan penguncian (lock), sehingga mengurangi potensi blokade antar transaksi. Sementara itu, transaksi yang melakukan pembaruan (update transactions) tetap mengikuti aturan 2PL untuk menjaga konsistensi dan serialisasi eksekusi. Dengan pemisahan antara transaksi pembaca dan penulis tersebut, MV2PL bertujuan untuk meningkatkan tingkat konkurensi tanpa mengorbankan integritas data maupun serializability. Karakteristik Utama MV2PL adalah sebagai berikut.

1. **Transaksi Read-Only**
   Dapat melakukan operasi pembacaan tanpa lock, dengan memanfaatkan snapshot dari basis data sesuai dengan timestamp transaksi. Hal ini meminimalkan konflik dengan transaksi penulis.
2. **Transaksi Update**
   Tetap mematuhi protokol two-phase locking, khususnya pada operasi penulisan, sehingga menjaga serialisasi eksekusi dan konsistensi perubahan data.
3. **Antrian Penungguan (Wait-Queue)**
   Jika suatu operasi tertahan karena item data sedang dikunci, operasi tersebut ditempatkan dalam antrian penungguan dan akan dieksekusi ketika lock dilepaskan.

4. **Pencegahan Deadlock - Wound-Wait**

   Menggunakan strategi wound-wait, yaitu transaksi yang memiliki timestamp lebih tua berhak "melukai" (wound) transaksi yang lebih muda dengan memaksa transaksi muda untuk rollback, sehingga deadlock dapat dicegah secara deterministik.

Berikut merupakan strategi pengembangan untuk algoritma MV2PL.

1. Transaction Initialization dengan Timestamp Counter

```
def begin_transaction(self, transaction: Transaction) -> AlgorithmResponse:
    if self.variant == MVCCVariant.MVTO:
        timestamp = transaction.transaction_id
    else:
        timestamp = self.ts_counter  # MV2PL menggunakan counter global

    trans_info = TransactionInfo(
        transaction=transaction,
        timestamp=timestamp,
        transaction_type=TransactionType.UPDATE
    )
    self.transaction_info[transaction.transaction_id] = trans_info
    transaction.set_status(TransactionStatus.Active)
```

Penjelasan:
Pada MV2PL, timestamp transaksi diperoleh dari global counter yang meningkat secara incremental (ts_counter). Pendekatan ini berbeda dengan MVTO yang menggunakan transaction ID sebagai timestamp statis. Penggunaan ts_counter pada MV2PL memastikan bahwa urutan serialisasi transaksi selalu terjaga secara konsisten, karena setiap transaksi selalu memperoleh timestamp yang lebih besar daripada transaksi sebelumnya, sesuai dengan urutan waktu aktual saat transaksi dimulai.

2. Read Operation - Lock-free untuk Read-only

```
def validate_read_mv2pl(self, transaction: Transaction, row: Row) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0)
        ]

    trans_info.read_set.add(object_id)

    if trans_info.transaction_type == TransactionType.READ_ONLY:
        suitable_version = None
        for version in sorted(self.data_versions[object_id],
                              key=lambda v: v.commit_ts or 0, reverse=True):
            if (version.commit_ts is not None and
                version.commit_ts < trans_info.timestamp):
```

```
                suitable_version = version
                break

        if not suitable_version:
            suitable_version = self.data_versions[object_id][0]

        return AlgorithmResponse(
            allowed=True,
            message=f"T{transaction.transaction_id} (read-only) reads {object_id}",
            value=suitable_version.value
        )

    if object_id in trans_info.locks_held:
        if trans_info.locks_held[object_id] == 'X':
            version = self.data_versions[object_id][-1]
            return AlgorithmResponse(
                allowed=True,
                message=f"T{transaction.transaction_id} (read-only) reads {object_id}",
                value=version.value
            )
```

Penjelasan:

- Transaksi Read-Only
  Operasi pembacaan oleh transaksi yang bersifat read-only dilakukan pada versi data yang telah committed dengan nilai commit_ts < TS(Ti). Mekanisme ini tidak memerlukan penguncian (lock), sehingga transaksi pembaca tidak menghambat transaksi lain.
- Transaksi Update
  Untuk transaksi yang melakukan pembaruan, operasi pembacaan membutuhkan shared lock pada item data. Apabila transaksi tersebut telah memiliki exclusive lock sebelumnya, maka pembacaan dapat dilakukan tanpa akuisisi lock tambahan.
- Pemilihan Versi (Version Selection)
  Sistem memilih versi data yang telah committed dan masih sesuai dengan timestamp transaksi. Versi yang dipilih adalah versi terbaru yang memiliki commit_ts lebih kecil atau sama dengan TS(Ti), sehingga konsistensi snapshot tetap terjaga.

3. Write Operation dengan Locking

```
def validate_write_mv2pl(self, transaction: Transaction, row: Row) -> MV2PLResponse:
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0,
                       creator_ts=None, commit_ts=0)
        ]

    existing_lock_holders = self._get_lock_holders_mv2pl(object_id)

    if existing_lock_holders and existing_lock_holders != {transaction.transaction_id}:
        can_proceed = True
        for holder_tid in existing_lock_holders:
            if holder_tid == transaction.transaction_id:
```

```
                continue
            holder_info = self.transaction_info.get(holder_tid)
            if holder_info:
                if trans_info.timestamp < holder_info.timestamp:
                    holder_info.transaction.set_status(TransactionStatus.Aborted)
                    if object_id in holder_info.locks_held:
                        del holder_info.locks_held[object_id]
                else:
                    can_proceed = False

        if not can_proceed:
            self.operation_queue.append({
                'type': 'write',
                'tid': transaction.transaction_id,
                'object_id': object_id,
                'row': row
            })
            self.blocked_transactions.add(transaction.transaction_id)

            return MV2PLResponse(
                allowed=False,
                message=f"T{transaction.transaction_id} blocked waiting for lock-X({object_id})",
                executed_ops=None
            )

    trans_info.locks_held[object_id] = 'X'
    trans_info.write_set.add(object_id)

    new_version = RowVersion(
        value=row.data.get('value', 0),
        version_id=len(self.data_versions[object_id]),
        read_ts=trans_info.timestamp,
        write_ts=trans_info.timestamp,
        creator_ts=trans_info.timestamp,
        commit_ts=None
    )
    self.data_versions[object_id].append(new_version)

    return MV2PLResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} writes {object_id}∞ (lock-X acquired)",
        executed_ops=None
    )
```

Penjelasan:

- Pemeriksaan Lock (Lock Check)
  Sistem terlebih dahulu memeriksa apakah terdapat transaksi lain yang sedang memegang lock pada item data yang akan ditulis.
- Penerapan Protokol Wound-Wait
  Jika terjadi konflik lock, mekanisme berikut digunakan untuk mencegah deadlock:
  - Apabila transaksi Ti memiliki timestamp lebih tua dibanding transaksi Tj yang memegang lock, maka Ti berhak untuk wound, yaitu memaksa Tj melakukan abort, sehingga Ti dapat memperoleh lock.
  - Apabila transaksi Ti lebih muda, maka Ti harus menunggu, dan permintaan lock tersebut ditempatkan dalam wait-queue.

- Akuisisi Exclusive Lock
  Jika transaksi diizinkan memperoleh lock, sistem memberikan exclusive lock pada item data, kemudian transaksi membuat versi baru yang belum di-commit sebagai hasil operasi penulisan.
- Manajemen Queue
  Operasi yang diblokir karena konflik lock disimpan dalam antrian penungguan (wait-queue) dan akan dieksekusi kembali ketika lock dilepaskan atau ketika transaksi yang memegang lock di-abort.

4. Commit dengan Lock Release & Queue Processing

```
def commit_transaction_mv2pl(self, transaction: Transaction) -> MV2PLResponse:
    trans_info = self.transaction_info[transaction.transaction_id]

    # Increment global timestamp counter
    self.ts_counter += 1
    commit_ts = self.ts_counter

    # Update commit_ts untuk semua versi yang dibuat oleh transaksi ini
    for obj_id in trans_info.write_set:
        if obj_id in self.data_versions:
            for version in self.data_versions[obj_id]:
                if (version.creator_ts == trans_info.timestamp and
                    version.commit_ts is None):
                    version.commit_ts = commit_ts

    # Release all locks
    released_locks = list(trans_info.locks_held.keys())
    trans_info.locks_held.clear()

    transaction.set_status(TransactionStatus.Committed)

    # Process operations dari queue yang sekarang bisa dieksekusi
    executed_ops = []
    if released_locks:
        executed_ops = self._process_queue_mv2pl(released_locks)

    return MV2PLResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} COMMIT (ts-counter={commit_ts})",
        executed_ops=executed_ops
    )

def _process_queue_mv2pl(self, released_objects: List[str]) -> List[Tuple]:
    executed_ops = []
    remaining_queue = []

    for op in self.operation_queue:
        if op['object_id'] in released_objects:
            # Try to execute this operation
            if op['type'] == 'write':
                result = self.validate_write_mv2pl(
                    self.transaction_info[op['tid']].transaction,
                    op['row']
```

```
                )
                if result.allowed:
                    executed_ops.append((
                        'write', op['tid'], result.message, result.value
                    ))
                else:
                    remaining_queue.append(op)
            elif op['type'] == 'read':
                result = self.validate_read_mv2pl(
                    self.transaction_info[op['tid']].transaction,
                    op['row']
                )
                if result.allowed:
                    executed_ops.append((
                        'read', op['tid'], result.message, result.value
                    ))
                else:
                    remaining_queue.append(op)
        else:
            remaining_queue.append(op)

    self.operation_queue = remaining_queue
    return executed_ops
```

Penjelasan:

Commit Logic

- Pemberian Commit Timestamp
  Pada saat transaksi melakukan commit, sistem meningkatkan nilai ts_counter secara incremental dan menetapkannya sebagai commit_ts untuk seluruh versi data yang dihasilkan oleh transaksi tersebut.
- Finalisasi Versi Data
  Semua versi yang sebelumnya berstatus uncommitted akan diperbarui menjadi committed dengan commit_ts yang telah ditetapkan. Langkah ini menandai bahwa perubahan yang dilakukan transaksi telah sah dan dapat diakses oleh transaksi lain.
- Pelepasan Seluruh Lock
  Setelah versi difinalisasi, semua lock yang dimiliki transaksi akan dibebaskan. Pelepasan lock ini memungkinkan operasi lain yang sebelumnya tertunda untuk dapat dilanjutkan.
- Pemrosesan Antrian (Queue Processing Trigger)
  Setelah lock dilepaskan, sistem melakukan pemeriksaan pada wait-queue untuk mengeksekusi kembali operasi-operasi yang sebelumnya diblokir karena konflik lock.

Queue Processing

- Iterasi Antrian
  Sistem menelusuri seluruh operasi yang terdapat dalam wait-queue.
- Pemeriksaan Ketersediaan Akses
  Untuk setiap operasi di dalam antrian, sistem memeriksa apakah item data yang dibutuhkan sudah tidak dikunci oleh transaksi lain.
- Eksekusi atau Penundaan Kembali
  Jika operasi sudah dapat dieksekusi (tidak ada konflik lock), maka sistem menjalankannya kembali. Jika konflik masih ada, operasi dikembalikan ke dalam antrian agar dievaluasi pada kesempatan berikutnya.

- Pengembalian Hasil
  Sistem mengumpulkan daftar operasi yang berhasil dijalankan selama proses ini, kemudian mengembalikannya sebagai hasil pemrosesan antrian.

c. Snapshot Isolation First-Comitter-Win

Snapshot Isolation (SI) merupakan mekanisme kontrol konkurensi yang memberikan setiap transaksi snapshot basis data yang konsisten pada saat transaksi dimulai. Seluruh operasi penulisan tidak langsung terlihat oleh transaksi lain, melainkan disimpan terlebih dahulu dalam buffer hingga transaksi melakukan commit. Snapshot Isolation dengan First-Committer-Wins (FCW) merupakan mekanisme pendeteksian konflik yang dilakukan pada saat transaksi melakukan commit, bukan pada waktu penulisan. Seluruh operasi penulisan disimpan terlebih dahulu dalam buffer tanpa memerlukan lock. Proses pendeteksian konflik baru terjadi pada tahap commit dengan memeriksa adanya konflik penulisan (write–write conflict). Dengan demikian, FCW memungkinkan tingkat konkurensi yang lebih tinggi karena tidak ada penguncian selama operasi penulisan berlangsung. Namun, mekanisme ini dapat menyebabkan pembatalan transaksi pada akhir eksekusi apabila konflik ditemukan saat commit. Berikut merupakan strategi pemgembangan yang dilakukan untuk algoritma Snapshot Isolation First-Comitter-Win.

1. Begin Transaction

```
def begin_transaction(self, transaction: Transaction) -> AlgorithmResponse:
    timestamp = self.ts_counter

    trans_info = TransactionInfo(
        transaction=transaction,
        timestamp=timestamp,
        transaction_type=TransactionType.UPDATE
    )
    self.transaction_info[transaction.transaction_id] = trans_info
    transaction.set_status(TransactionStatus.Active)
```

Penjelasan:
Saat transaksi BEGIN, sistem mencatat: snapshot_ts = current_global_timestamp

Konsekuensinya:

- Transaksi hanya dapat melihat versi data yang sudah committed sebelum atau tepat saat snapshot_ts.
- Versi commit setelah transaksi mulai tidak akan terlihat (isolasi snapshot).
- Versi lama bersifat immutable, sehingga read tidak memblok write atau sebaliknya (no read locks).

2. Read dari Snapshot

```
def validate_read_snapshot(self, transaction: Transaction, row: Row) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0,
                       creator_ts=0, commit_ts=0)
        ]

    trans_info.read_set.add(object_id)

    suitable_version = None
    for version in sorted(self.data_versions[object_id],
                          key=lambda v: v.commit_ts or 0, reverse=True):
        if (version.commit_ts is not None and
            version.commit_ts <= trans_info.timestamp):
            suitable_version = version
            break

    if not suitable_version:
        suitable_version = self.data_versions[object_id][0]

    return AlgorithmResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} reads {object_id} (snapshot)",
        value=suitable_version.value
    )
```

Penjelasan:
Untuk setiap operasi baca, database akan:

- Mencari versi data terbaru yang memiliki commit_ts ≤ snapshot_ts.
- Mengabaikan versi yang belum commit atau commit setelah transaksi mulai.
- Langsung dapat dibaca tanpa blocking.

3. Write-Buffer tanpa Lock

```
def validate_write_snapshot_fcw(self, transaction: Transaction, row: Row) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]
```

```
    object_id = row.object_id

    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0,
                       creator_ts=0, commit_ts=0)
        ]

    trans_info.write_set.add(object_id)

    trans_info.buffered_writes[object_id] = row.data.get('value', 0)

    return AlgorithmResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} writes {object_id} (buffered)"
    )
```

Penjelasan:
First-Committer-Wins (FCW) merupakan aturan deteksi konflik pada Snapshot Isolation yang dilakukan pada waktu commit, bukan saat write. Seluruh operasi write selama transaksi hanya disimpan di buffer tanpa penguncian objek saat penulisan.

4. Commit dengan Write-Write Conflict Detection

```
def commit_transaction_snapshot(self, transaction: Transaction) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]

    if self.isolation_policy == IsolationPolicy.FIRST_COMMITTER_WIN:
        for obj_id in trans_info.write_set:
            if obj_id in self.data_versions:
                for version in self.data_versions[obj_id]:
                    if (version.commit_ts is not None and
                        version.commit_ts > trans_info.timestamp):
                        transaction.set_status(TransactionStatus.Aborted)
                        return AlgorithmResponse(
                            allowed=False,
                            message=f"T{transaction.transaction_id} ABORTED: "
                                    f"write-write conflict"
                        )

    self.ts_counter += 1
    commit_ts = self.ts_counter

    for obj_id, value in trans_info.buffered_writes.items():
        new_version = RowVersion(
            value=value,
            version_id=len(self.data_versions[obj_id]),
            read_ts=commit_ts,
            write_ts=commit_ts,
            creator_ts=trans_info.timestamp,
            commit_ts=commit_ts
        )
        self.data_versions[obj_id].append(new_version)
```

```
    trans_info.buffered_writes.clear()
    transaction.set_status(TransactionStatus.Committed)

    return AlgorithmResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} COMMIT (ts-counter={commit_ts})"
    )
```

Penjelasan:
Saat transaksi akan COMMIT, sistem melakukan prosedur berikut:

1. Iterasi Write Set
   Periksa setiap objek yang ditulis oleh transaksi.
2. Cek Versi yang Sudah Commit
   Cari versi objek yang memiliki:
   commit_ts > snapshot_ts
   Artinya ada transaksi lain sudah commit lebih dahulu pada objek yang sama.
3. Conflict Detected → ABORT
   Karena transaksi lain sudah "menang" commit pada objek tersebut, transaksi ini gagal.
4. Jika Tidak Ada Konflik → Apply Writes
   - Buat versi baru untuk seluruh write di-buffer.
   - Beri commit timestamp yang baru.
   - Writes menjadi visible untuk transaksi lain.

d. Snapshot Isolation First-Updater-Win

Snapshot Isolation menggunakan pendekatan First-Updater-Wins (FUW) untuk mendeteksi konflik secara lebih awal, yaitu pada saat operasi penulisan dilakukan. Dengan demikian, Snapshot Isolation dapat mencegah write–write conflict sebelum transaksi berlanjut lebih jauh dalam eksekusinya. Berikut merupakan strategi pemgembangan yang dilakukan untuk algoritma Snapshot Isolation First-Updater-Win.

1. Begin Transaction dengan Snapshot Timestamp

```
def begin_transaction(self, transaction: Transaction) -> AlgorithmResponse:
    timestamp = self.ts_counter

    trans_info = TransactionInfo(
        transaction=transaction,
        timestamp=timestamp,
        transaction_type=TransactionType.UPDATE
    )
    self.transaction_info[transaction.transaction_id] = trans_info
    transaction.set_status(TransactionStatus.Active)
```

Penjelasan:
Saat transaksi BEGIN, sistem mencatat: snapshot_ts = current_global_timestamp
Konsekuensinya:

- Transaksi hanya dapat melihat versi data yang sudah committed sebelum atau tepat saat snapshot_ts.
- Versi commit setelah transaksi mulai tidak akan terlihat (isolasi snapshot).
- Versi lama bersifat immutable, sehingga read tidak memblok write atau sebaliknya (no read locks).

2. Read dari Snapshot

```
def validate_read_snapshot(self, transaction: Transaction, row: Row) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    # Inisialisasi versi jika belum ada
    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0,
                       creator_ts=0, commit_ts=0)
        ]

    trans_info.read_set.add(object_id)

    # Pilih versi dari snapshot (commit_ts <= snapshot_ts)
    suitable_version = None
    for version in sorted(self.data_versions[object_id],
                          key=lambda v: v.commit_ts or 0, reverse=True):
        if (version.commit_ts is not None and
            version.commit_ts <= trans_info.timestamp):
            suitable_version = version
            break

    if not suitable_version:
        suitable_version = self.data_versions[object_id][0]

    return AlgorithmResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} reads {object_id} (snapshot)",
        value=suitable_version.value
    )
```

Penjelasan:
Untuk setiap operasi baca, database akan:

- Mencari versi data terbaru yang memiliki commit_ts ≤ snapshot_ts.
- Mengabaikan versi yang belum commit atau commit setelah transaksi mulai.
- Langsung dapat dibaca tanpa blocking.

3. Write dengan Exclusive Lock (First-Updater-Wins)

```
def validate_write_snapshot_fuw(self, transaction: Transaction, row: Row) -> AlgorithmResponse:
```

```
    trans_info = self.transaction_info[transaction.transaction_id]
    object_id = row.object_id

    # Inisialisasi versi jika belum ada
    if object_id not in self.data_versions:
        self.data_versions[object_id] = [
            RowVersion(value=0, version_id=0, read_ts=0, write_ts=0,
                       creator_ts=0, commit_ts=0)
        ]

    # First-Updater-Wins: Check apakah ada transaksi lain yang sudah punya exclusive lock
    for tid, other_info in self.transaction_info.items():
        if tid == transaction.transaction_id:
            continue
        if object_id in other_info.exclusive_locks:
            # Transaksi lain sudah punya exclusive lock → ABORT
            transaction.set_status(TransactionStatus.Aborted)
            return AlgorithmResponse(
                allowed=False,
                message=f"T{transaction.transaction_id} ABORTED: "
                        f"exclusive lock conflict on {object_id}"
            )

    # Acquire exclusive lock
    trans_info.exclusive_locks.add(object_id)
    trans_info.write_set.add(object_id)

    # Buffer write (belum apply ke data_versions)
    trans_info.buffered_writes[object_id] = row.data.get('value', 0)

    return AlgorithmResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} writes {object_id} (buffered)"
    )
```

Penjelasan:
Snapshot Isolation mencegah write-write conflict melalui aturan First-Updater Win:

1. Saat transaksi melakukan write pertama pada sebuah objek, sistem mengecek apakah objek itu sudah di-lock updater lain.
2. Jika ya, Immediate Abort, transaksi gagal tanpa menunggu.
3. Jika tidak, transaksi mendapat exclusive lock pada objek tersebut.
4. Perubahan tidak langsung ditulis ke database → di-buffer terlebih dahulu.

4. Commit - Apply Buffered Writes

```
def commit_transaction_snapshot(self, transaction: Transaction) -> AlgorithmResponse:
    trans_info = self.transaction_info[transaction.transaction_id]

    self.ts_counter += 1
    commit_ts = self.ts_counter

    for obj_id, value in trans_info.buffered_writes.items():
        new_version = RowVersion(
            value=value,
            version_id=len(self.data_versions[obj_id]),
```

```
            read_ts=commit_ts,
            write_ts=commit_ts,
            creator_ts=trans_info.timestamp,
            commit_ts=commit_ts
        )
        self.data_versions[obj_id].append(new_version)

    trans_info.exclusive_locks.clear()
    trans_info.buffered_writes.clear()

    transaction.set_status(TransactionStatus.Committed)

    return AlgorithmResponse(
        allowed=True,
        message=f"T{transaction.transaction_id} COMMIT (ts-counter={commit_ts})"
    )
```

Penjelasan:
Jika transaksi berhasil sampai commit:
1. global_ts_counter += 1
   Berikan commit timestamp baru untuk keseluruhan transaksi.
2. Untuk setiap write yang di-buffer:
   Dibuat versi baru dari objek: (value, commit_ts = global_ts_counter).
3. Exclusive locks di-release.
4. Status transaksi = Committed.

3. Hubungan Antar Komponen

CCManager merupakan komponen yang akan dipanggil menjadi singleton pada QueryProcessor, dengan demikian seluruh transaksi hanya akan melewati satu instance CCM untuk memastikan konkurensi dari transaksi. Pada Query Processor CCM diimplementasikan menggunakan adapter pattern. Berikut merupakan cuplikan implementasinya.

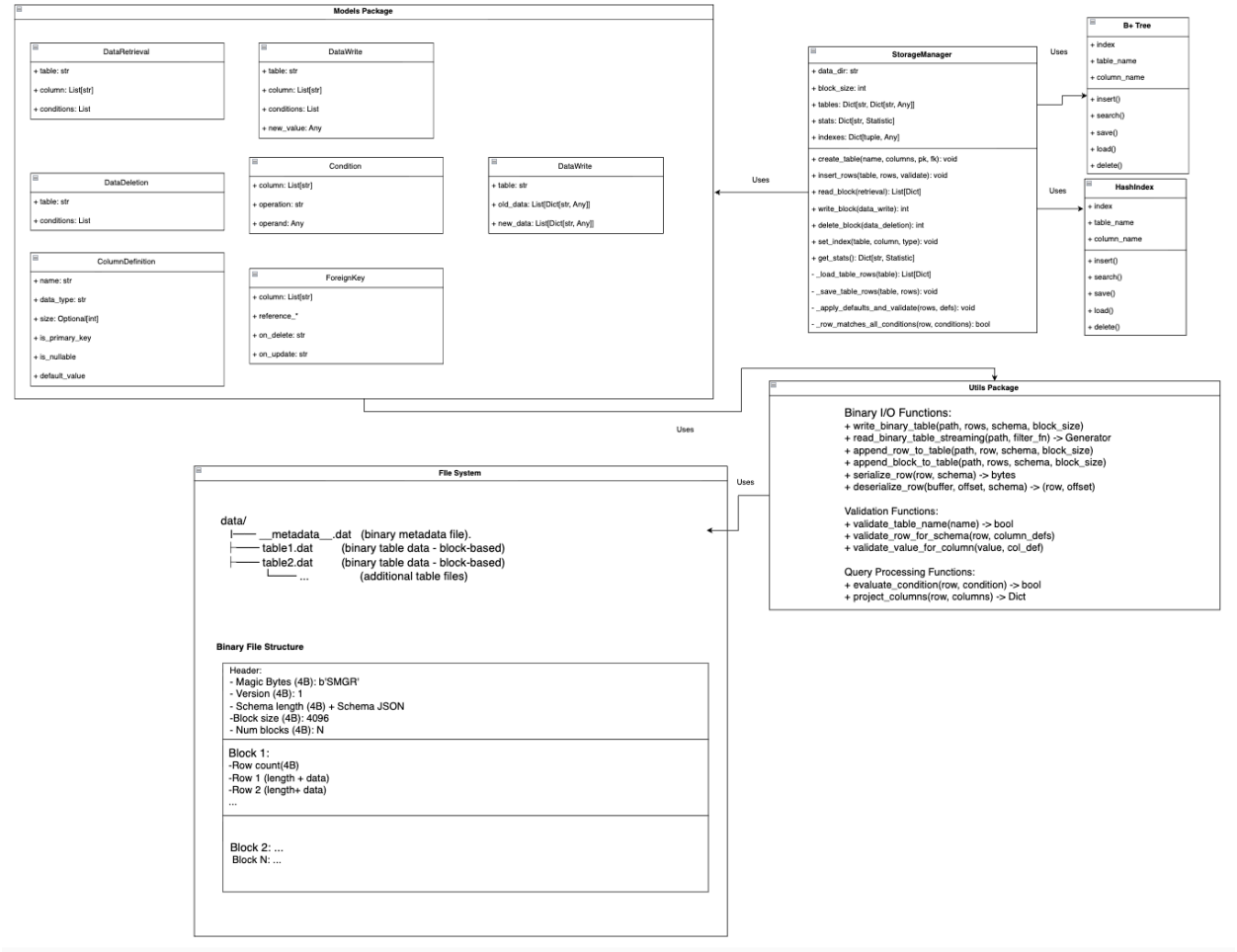
```
# query_processor/adapter_ccm.py
class AdapterCCM:
    def __init__(self, algorithm: AlgorithmType):
        self.ccm = CCManager(algorithm=algorithm, log_file="cc_log.txt")

    def begin_transaction(self) -> int:
        return self.ccm.begin_transaction()

    def validate_action(self, transaction_id: int, table_name: str,
                        action_type: str) -> Response:
        row = Row(object_id=table_name, table_name=table_name, data={})
        action = ActionType.READ if action_type == "read" else ActionType.WRITE
        return self.ccm.validate_object(row, transaction_id, action)
```

```
    def commit_transaction(self, transaction_id: int):
        self.ccm.end_transaction(transaction_id)

    def abort_transaction(self, transaction_id: int):
        self.ccm.abort_transaction(transaction_id)
```

CCM tidak berhubungan dengan komponen lain secara langsung kecuali dengan Query Processor karena seluruh integrasi terdapat pada Query Processor. Walau demikian, tetap terdapat alur integrasi yang terjadi antar komponen. Misalnya, untuk mengambil suatu data ke Storage Manage maka Query Processor perlu untuk melakukan validasi aksi dulu sebelum dapat melanjutkan pengambilan data ke Storage Manager, ketika validasi gagal maka transaksi akan di-*rollback*. Kemudian CCM juga independen terhadap Failure Recovery Manager (FRM) karena pemanggilan fungsi nya tergabung pada Query Processor, walaupun terdapat fungsi yang mirip yaitu logging namun fungsi tersebut memiliki perbedaan dengan CCM berfokus pada logging internal, sedangkan logging FRM berfungsi untuk melakukan ulang transaksi yang gagal.

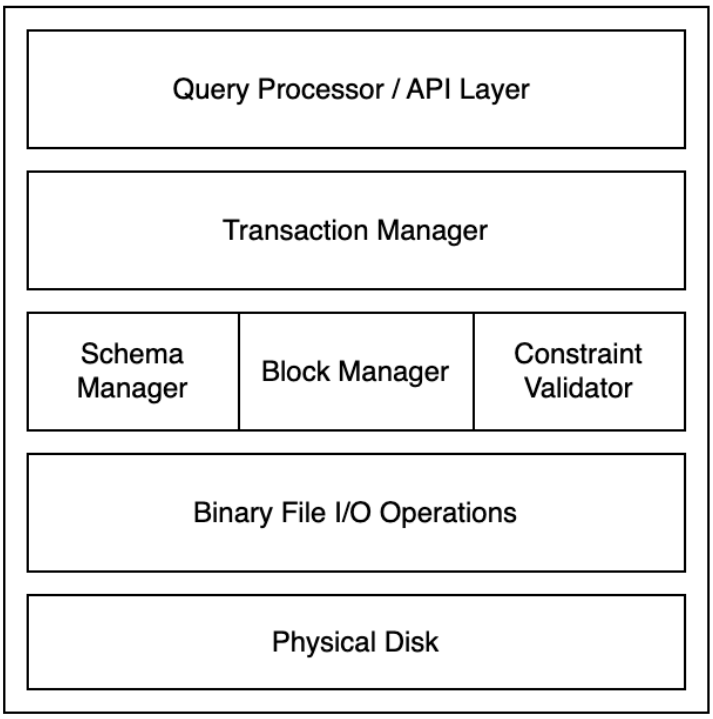
4. Storage Manager

Storage Manager adalah salah satu komponen fundamental dalam sistem mDBMS yang bertanggung jawab dalam mengelola penyimpanan data secara persisten di disk. Komponen ini mengimplementasikan mekanisme *block-based* storage dengan optimasi *streaming* untuk efisiensi memori dan performa.



Gambar 4.1 Visualisasi Class Diagram Storage Manager

Storage Manager dirancang sebagai *layer* abstraksi antara komponen tingkat tinggi seperti Query Processor dan *physical storage*, sehingga komponen lain tidak perlu mengetahui detail implementasi penyimpanan *binary file*.



Gambar 4.2 Visualisasi Layer Abstraksi

Tanggung jawab utama Storage Manager mencakup beberapa hal berikut:

1. **Persistensi Data**

   Menyimpan dan membaca data dari disk dalam *format* binary yang efisien.

   Format Binary:

   - Header Block: Metadata tabel (schema, constraints)
   - Data Blocks: Data rows dalam *fixed-size blocks*
   - Index Blocks (optional): B-tree atau Hash indexes untuk *fast lookup*

2. **Schema management**

   Mengelola *metadata* tabel termasuk kolom, tipe data, dan *constraints*

3. **Block management**

   Mengelola data dalam fixed-size blocks untuk efisiensi I/O

4. **Constraint Enforcement**

   Validasi constraints: Primary Key, Foreign Key, Unique, Not Null

5. **Data Model**

Integrasi Storage Manager dengan komponen lainnya:

1. Query Processor

```
// Class QueryExecution menggunakan Storage Manager melalui AdapterStorage

class QueryExecution:
    def __init__(self, storage_adapter=None, storage_manager=None):
        # Initialize storage adapter
        if storage_manager:
            self.storage_adapter = AdapterStorage(storage_manager=storage_manager)
            self.storage_manager = storage_manager

    def execute_project(self, query_tree, transaction_id):
        # Panggil storage manager untuk SELECT
        data_retrieval = DataRetrieval(...)
        source_data = self.storage_manager.read_block(data_retrieval)

    def execute_filter(self, query_tree, transaction_id):
        # Push down WHERE ke storage manager
        conditions = self.condition_tree_to_conditions(condition_tree)
        filtered_data = self.storage_manager.read_block(data_retrieval)

    def execute_insert(self, query_tree, transaction_id):
        # Buffer insert atau write langsung ke storage
        self.storage_adapter.write_data(...)

    def execute_update(self, query_tree, transaction_id):
        # READ-MODIFY-WRITE pattern
        matching_rows = self.storage_adapter.read_data(...)
        # Update logic...
        self.storage_adapter.write_data(...)
```

Pola Integrasi:

- QueryExecution → AdapterStorage → StorageManager
- AdapterStorage adalah wrapper/adapter untuk Storage Manager
- Query Processor menggunakan adapter agar tidak directly coupled ke Storage Manager

2. Query Optimizer

```
// Query Optimizer memberikan optimized query tree ke Query Execution

class AdapterOptimizer:
    def parse_optimized_query(self, sql_query):
        # Parse SQL → Optimize → Return QueryTree
        parsed = parser.parse(sql_query)
        optimized = optimizer.optimize(parsed)
        return optimized

# Di QueryExecution:
def _get_execution_method(self, node: QueryTree):
    """Delegate ke optimizer adapter untuk execution method"""
    return self.optimizer_adapter.get_execution_method(node)
```

3. Concurrency Control Manager (CCM)

```
// CCM mengontrol akses concurrent ke Storage Manager

class AdapterCCM:
    def validate_action(self, transaction_id, table_name, action):
        # Check lock compatibility
        # Return allowed/denied

# Di QueryExecution:
def execute_project(self, query_tree, transaction_id):
    # Validate READ access
    if self.ccm_adapter:
        response = self.ccm_adapter.validate_action(
            transaction_id, table_name, 'read'
        )
        if not response.allowed:
            # Check if aborted (Wait-Die deadlock prevention)
            if "aborted" in response.message.lower():
                raise TransactionAbortedException(...)
            return []

    # Proceed to storage manager...
    data = self.storage_manager.read_block(...)
```

Pola Integrasi:

- CCM di-*check* sebelum akses Storage Manager
- CCM mengelola *locks* untuk isolasi transaksi
- Jika CCM *deny* → tidak boleh akses Storage Manager

4. Failure Recovery Manager (FRM)

```
// FRM logging untuk recovery
```

```
class AdapterFRM:
    def log_write(self, transaction_id, query, table_name,
                  old_data, new_data):
        # Write to WAL (Write-Ahead Log)

# Di QueryExecution:
def execute_update(self, query_tree, transaction_id):
    # Read data
    matching_rows = self.storage_adapter.read_data(...)

    # Log to FRM BEFORE writing
    if self.frm_adapter and transaction_id:
        self.frm_adapter.log_write(
            transaction_id=transaction_id,
            query=f"UPDATE {table_name}",
            table_name=table_name,
            old_data=matching_rows,  # For rollback
            new_data=rows_to_update
        )

    # Then write to storage
    self.storage_adapter.write_data(...)
```

Pola Integrasi:

- FRM log sebelum *write* ke Storage Manager (WAL *principle*)
- Log digunakan untuk *recovery* jika *crash*
- FRM tidak *block* Storage Manager, hanya *logging*

5. Transaction Buffer

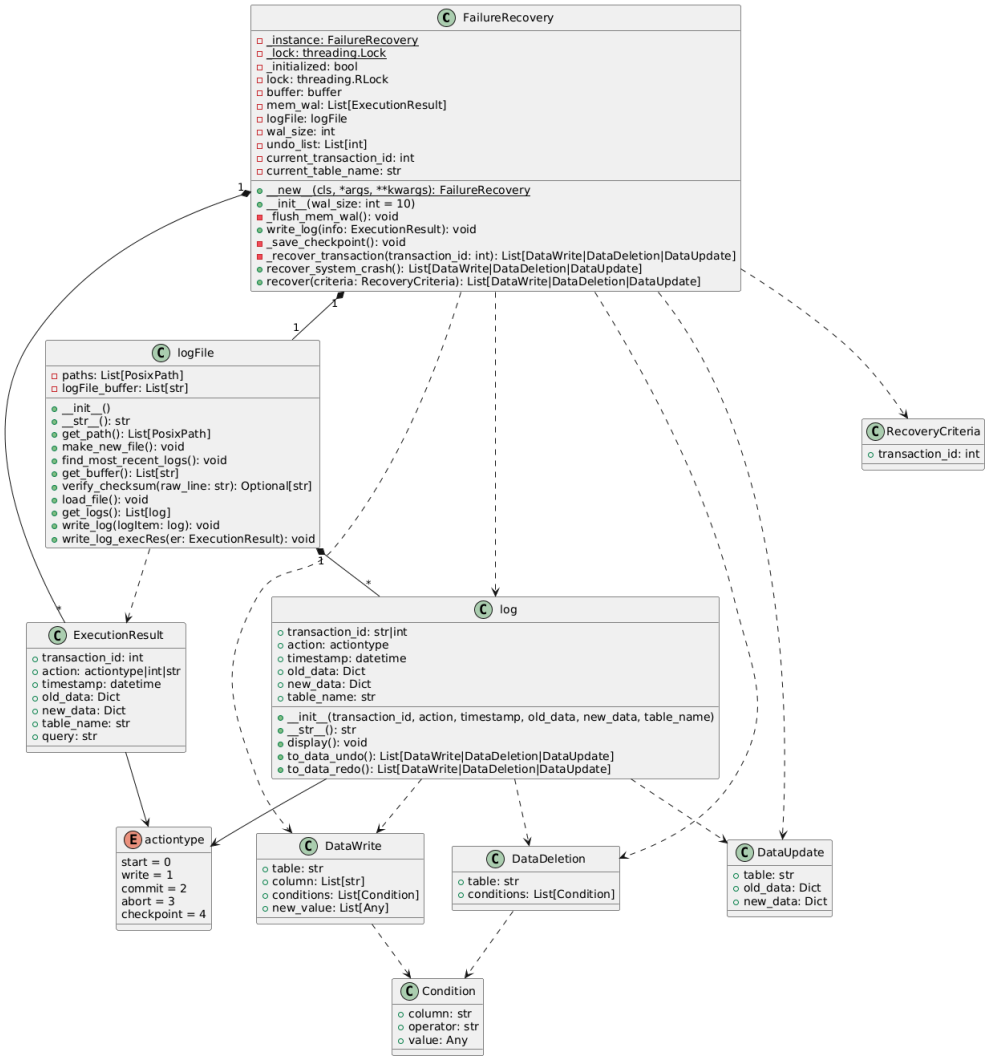
```
// Transaction Buffer untuk isolasi

class TransactionBuffer:
    def buffer_insert(self, transaction_id, table_name, row_data):
        # Buffer insert, tidak langsung ke storage

    def buffer_update(self, transaction_id, table_name,
                      old_data, new_data, conditions):
        # Buffer update

    def buffer_delete(self, transaction_id, table_name,
                      row_data, conditions):
        # Buffer delete

# Di QueryExecution:
def execute_insert(self, query_tree, transaction_id):
    new_row = dict(zip(columns, values))

    if transaction_id:
        # Buffer instead of direct write
```

```
            self.transaction_buffer.buffer_insert(
                transaction_id, table_name, new_row
            )
            return 1
        else:
            # No transaction - write directly to storage
            self.storage_adapter.write_data(...)
```

Pola Integrasi:

- Buffer menyimpan *uncommitted* changesSaat *COMMIT* → *flush buffer* ke Storage Manager
- Saat *ROLLBACK* → discard buffer
- Ini implementasi *Read Committed isolation level*

5. Failure Recovery



FRM_Diagram.png

Gambar 5.1 Visualisasi Class Diagram Failure Recovery Manager

Failure Recovery Manager atau FRM berfungsi untuk mengatasi error sebab kegagalan pada transaksi di DBMS, sehingga FRM berperan penting untuk menjaga konsistensi data pada semua user yang terkait. Failure Recovery Manager berhubungan dengan executionResult dari Query Processor untuk proses logging file. Berikut merupakan kelas yang terlibat dalam FRM:

- **FailureRecovery**

  Class FailureRecovery merupakan sebuah singleton class yang digunakan untuk segala operasi failure recovery. Singleton class ini diadaptasi dengan menggunakan class AdapterFRM, yang berfungsi untuk mengintegrasikan singleton instance FailureRecovery dengan class lainnya. FailureRecovery memiliki metode write_log untuk mencatat log pada sebuah file melalui kelas logFile, save_checkpoint untuk menambahkan entri checkpoint pada log, recover_transaction untuk melakukan recovery saat transaction fail abruptly, dan fungsi recover_system_crash untuk recovery saat sistem mengalami kegagalan. Cara failure recovery bekerja secara keseluruhan yaitu dengan menyimpan log dari executionResult pada sebuah list dalam memori yang kemudian akan di-write pada file ketika fungsi write_log terpanggil atau ketika tercapai batasan limit memory untuk list of "log"s tersebut..

- **Log**

  Class Log merepresentasikan sebuah objek log yang mencakup atribut - atribut sebagai berikut:

```
transaction_id: str | int,
action: actiontype,
timestamp: datetime,
old_data: Dict = None,
new_data: Dict = None,
table_name : str = None
```

  Log yang dideklarasikan akan pertama tersimpan pada memori dalam sebuah list lalu ditulis pada sebuah file yang mengandung log tersebut, log yang tertulis pada file akan memiliki semua informasi pada atribut tercantum.

- **LogFile**

  Class logFile bertanggung jawab untuk menulis list of "log"s yang ada pada memori pada sebuah file yang diinisialisasi ketika sebuah objek logFile

terinisialisasi. Class logFile akan membuat dua instansi dari file ".log" dengan format sebagai berikut:

```
curr_time = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
filename_1 = f"logfile_{curr_time}_1.log"
filename_2 = f"logfile_{curr_time}_2.log"
```

Class logFile memiliki metode write_log untuk menuliskan objek log pada file yang telah diinisialisasi, dengan menambahkan checksum di belakang setiap log yang akan ditulis sebagai berikut:

```
data = str(logItem)
checksum = hashlib.sha256(data.encode('utf-8')).hexdigest(

# File Related Operations
f.write(f"{checksum}|{data}\n")
```

Checksum yang ditambahkan akan diverifikasi oleh class logFile dengan sebuah metode, dan apabila checksum berhasil maka data akan dikembalikan, apabila tidak maka "None" akan dikembalikan:

```
def verify_checksum(self, raw_line: str) -> Optional[str]:
    line = raw_line.rstrip("\n")
    if line == "":
        return None

    try:
        checksum, data = line.split("|", 1)
    except ValueError:
        return None

    new_checksum = hashlib.sha256(data.encode('utf-8')).hexdigest()
    if checksum != new_checksum:
        return None
    return data
```

Maybe nanti jelasin integrasi?

Part III. Experiment

1. Query Processor

a. Test Query Processor

Description	Melakukan eksekusi terhadap beberapa jenis *query* yang dapat diolah oleh *query processor* berdasarkan spek.
Goal	Memastikan *query* dapat diproses selayaknya sebuah DBMS.
Method	Untuk setiap *query* test, melakukan pembuatan data temp untuk menampung relasi, lalu untuk setiap *query*, dibersihkan dan dibangun ulang relasi-relasi tersebut. Create table dan insert into didahulukan agar dipanggil untuk setiap test.
Success Criterion	Query dapat diproses dan memberikan hasil yang diharapkan berdasarkan data yang terdapat pada relasi.
Input	`=== CREATE TABLE ===` `Query 1:` `CREATE TABLE departments (` `dept_id INTEGER PRIMARY KEY,` `dept_name VARCHAR(50),` `budget INTEGER` `)` `Query 2:` `CREATE TABLE employees (` `emp_id INTEGER PRIMARY KEY,` `emp_name VARCHAR(50),` `dept_id INTEGER,` `salary INTEGER,` `hire_date VARCHAR(20)` `)` `Query 3:` `CREATE TABLE projects (` `project_id INTEGER PRIMARY KEY,` `project_name VARCHAR(50),` `dept_id INTEGER,` `budget INTEGER` `)` `Query 4:` `CREATE TABLE tasks (` `task_id INTEGER PRIMARY KEY,` `task_name VARCHAR(50),` `project_id INTEGER,` `status VARCHAR(20)` `)` `=== INSERT INTO ===` `Query 5:` `INSERT INTO departments (dept_id, dept_name, budget) VALUES (1, 'Engineering', 100000);` `INSERT INTO departments (dept_id, dept_name, budget) VALUES (2, 'Sales', 80000);` `INSERT INTO departments (dept_id, dept_name, budget) VALUES (3, 'HR', 60000);`

```
=== SELECT ===
        dept_data = [
            (1, 'Engineering', 100000),
            (2, 'Sales', 80000),
            (3, 'HR', 60000)
        ]

        emp_data = [
            (1, 'Alice', 1, 60000, '2024-01-15'),
            (2, 'Bob', 1, 75000, '2023-06-20'),
            (3, 'Carol', 2, 55000, '2024-03-10'),
            (4, 'Dave', 2, 82000, '2022-11-05'),
            (5, 'Eve', 3, 50000, '2024-02-28')
        ]



Query 6:
SELECT * FROM departments

Query 7:
SELECT emp_id, emp_name FROM employees

Query 8:
SELECT * FROM employees WHERE salary > 60000

Query 9:
SELECT * FROM employees WHERE dept_id = 1

Query 10:
SELECT * FROM employees WHERE salary > 55000 AND dept_id = 2

Query 11:
SELECT emp_name, salary FROM employees WHERE salary > 50000 AND salary < 80000

Query 12:
SELECT * FROM employees WHERE dept_id = 1 OR dept_id = 3

Query 13:
SELECT emp_name, dept_id, salary FROM employees WHERE (dept_id = 1 AND salary > 70000) OR dept_id = 2

Query 14:
SELECT emp_name, dept_id, salary FROM employees WHERE (dept_id = 1 OR dept_id = 2) AND salary > 60000

Query 15:
SELECT * FROM employees WHERE salary = 60000 OR salary = 75000 OR salary = 50000

Query 16:
SELECT emp_name, dept_id, salary FROM employees WHERE dept_id = 1 AND salary > 60000 OR dept_id = 3


=== JOIN ===

        dept_data = [
            (1, 'Engineering', 100000),
            (2, 'Sales', 80000),
            (3, 'HR', 60000)
        ]
```

```
        emp_data = [
            (1, 'Alice', 1, 60000, '2024-01-15'),
            (2, 'Bob', 1, 75000, '2023-06-20'),
            (3, 'Carol', 2, 55000, '2024-03-10')
        ]


Query 17:
SELECT * FROM employees JOIN departments ON employees.dept_id = departments.dept_id

Query 18:
SELECT * FROM employees NATURAL JOIN departments

Query 19:
SELECT emp_name, dept_name FROM employees JOIN departments ON employees.dept_id = departments.dept_id

Query 20:
SELECT emp_name, dept_name FROM employees, departments

Query 21:
SELECT emp_name, budget FROM employees, departments

Query 22:
SELECT emp_name, dept_name FROM employees, departments WHERE employees.salary > 60000 AND
departments.budget > 70000

=== UPDATE ===

        emp_data = [
            (1, 'Alice', 1, 60000, '2024-01-15'),
            (2, 'Bob', 1, 75000, '2023-06-20'),
            (3, 'Carol', 2, 55000, '2024-03-10')
        ]

Query 23:
UPDATE employees SET salary = 65000 WHERE emp_id = 1

Query 24:
UPDATE employees SET salary = salary + 5000 WHERE dept_id = 1

=== DELETE ===

        task_data = [
            (1, 'Design UI', 1, 'completed'),
            (2, 'Implement API', 1, 'in_progress'),
            (3, 'Write Tests', 1, 'completed'),
            (4, 'Deploy', 2, 'pending')
        ]

Query 25:
DELETE FROM tasks WHERE status = 'completed'

=== LIMIT ===

        for i in range(1, 11):
            self.execute_query(f"INSERT INTO employees (emp_id, emp_name, dept_id, salary, hire_date)
VALUES ({i}, 'Employee{i}', 1, {50000 + i*1000}, '2024-01-01')")
```

```
Query 26:
SELECT * FROM employees WHERE salary > 52000 LIMIT 3

=== ORDER BY ===
        emp_data = [
            (1, 'Alice', 1, 60000, '2024-01-15'),
            (2, 'Bob', 1, 75000, '2023-06-20'),
            (3, 'Carol', 2, 55000, '2024-03-10'),
            (4, 'Dave', 2, 82000, '2022-11-05'),
            (5, 'Eve', 3, 50000, '2024-02-28')
        ]

Query 27:
SELECT * FROM employees WHERE dept_id = 1 ORDER BY salary DESC

=== ALIAS ===
        self.execute_query("INSERT INTO departments (dept_id, dept_name, budget) VALUES (1,
'Engineering', 100000)")
        self.execute_query("INSERT INTO employees (emp_id, emp_name, dept_id, salary, hire_date) VALUES
(1, 'Alice', 1, 60000, '2024-01-15')")

Query 28:
SELECT e.emp_name, d.dept_name FROM employees e JOIN departments d ON e.dept_id = d.dept_id

=== DROP TABLE ===

Query 29:
        CREATE TABLE temp_table (
            id INTEGER PRIMARY KEY,
            name VARCHAR(50)
        )
       DROP TABLE temp_table;

Query 30:
        CREATE TABLE temp_data (
            id INTEGER PRIMARY KEY,
            value INTEGER
        )

=== SUBQUERY ===
dept_data = [(1, 'Engineering', 100000), (2, 'Sales', 80000), (3, 'HR', 60000)]

        emp_data = [
            (1, 'Alice', 1, 75000),
            (2, 'Bob', 1, 65000),
            (3, 'Charlie', 2, 60000),
            (4, 'Diana', 2, 55000),
            (5, 'Eve', 99, 50000)
        ]

Query 31:
        SELECT emp_name, salary
        FROM employees
        WHERE dept_id IN (SELECT dept_id FROM departments WHERE budget > 70000)

Query 32:
        SELECT emp_name
        FROM employees
        WHERE dept_id IN (
```

	`` SELECT dept_id FROM departments WHERE dept_id IN (SELECT dept_id FROM departments WHERE budget > 70000)) ```
Expected Output	CREATE TABLE: - assert semua tables dalam storage_manager.tables INSERT INTO: - assert 3 rows ketika meng-query SELECT SELECT: - successful query, mengembalikan rows, assert berdasarkan conditions JOIN: - successful query, assert berdasarkan ekspektasi UPDATE: - successful query, assert affected rows DELETE: - successful query, assert affected rows LIMIT: - successful query, assert jumlah rows ORDER BY: - assert successful query ALIAS: - assert successful query dan tidak kosong DROP TABLE: - assert table yang ada dalam storage_manager.tables SUBQUERY: - successful query, mengembalikan rows, assert berdasarkan conditions
Results	SUCCESS

b. Test Concurrent Transactions

Description	Menggunakan Client untuk mengecek implementasi locking sudah cocok untuk digunakan atau belum.

Goal	Memastikan algoritma *deadlock detection* dan *prevention* bekerja dengan baik seiring berjalannya query.
Method	Menjalankan beberapa klien bersamaan dalam thread berbeda.
Success Criterion	Jika transaksi tidak ada deadlock dan transaksi yang di-abort sesuai.
Input	`queries_1 = [` `  "BEGIN TRANSACTION",` `  "UPDATE accounts SET balance = 1100 WHERE id = 1",` `  "COMMIT"` `]` `queries_2 = [` `  "BEGIN TRANSACTION",` `  "SELECT * FROM accounts WHERE id = 1",` `  "COMMIT"` `]` `thread1 = threading.Thread(target=self._execute_transaction, args=(1, queries_1))` `thread2 = threading.Thread(target=self._execute_transaction, args=(2, queries_2, 0.2))` `thread1.start()` `thread2.start()` `thread1.join()` `thread2.join()`
Expected Output	thread2 abort, thread1 commits
Results	SUCCESS

c. Test Transaction Buffer dan Recovery

Description	Melakukan test terhadap integrasi FRM dengan BufferTransaction yang terdapat di Query Processor.
Goal	Memastikan *logging*, *checkpointing*, dan *recovery* cocok dengan implementasi BufferTransaction
Method	Memasang ukuran Write Ahead Log yang kecil pada FRM, lalu dalam satu transaksi mengulangi query INSERT pada tabel kosong hingga terdapat checkpoint, lalu abort.

Success Criterion	Tabel kosong
Input	`num_inserts = 15` `for i in range(num_inserts):` `  insert_result = self.execute_query(f"INSERT INTO employees (emp_id, emp_name, salary) VALUES ({i}, 'Employee_{i}', {50000 + i * 1000})")` `abort_result = self.execute_query("ABORT")`
Expected Output	assert empty table
Results	SUCCESS

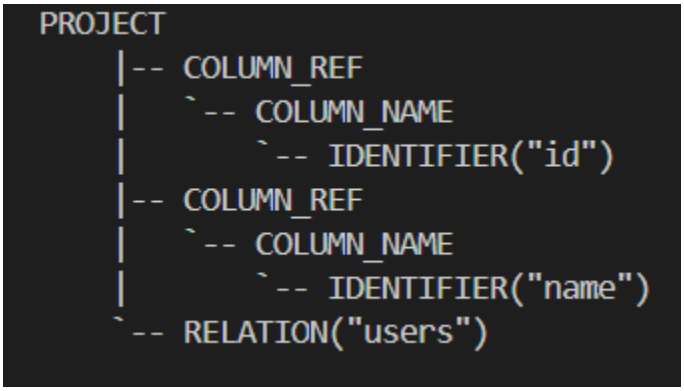
2. Query Optimizer

a. Demo Optimization

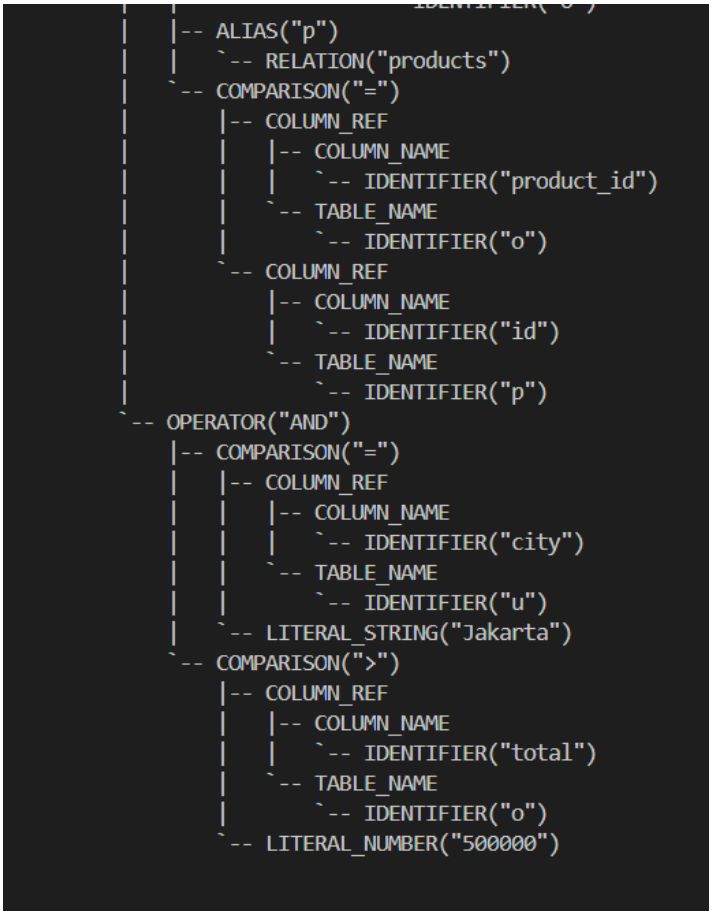
Description	Menguji secara umum parse_query, optimize dengan dan tanpa GA untuk mengecek apakah ada perbedaan cost dan memastikan validitasnya
Goal	Memastikan semua berjalan dengan benar dan hasil query optimized tetap valid
Method	Menjalankan parse_query, optimize_query, get_cost dan check_query. Dijalankan dengan command: `python -m query_optimizer.demo 10`
Success Criterion	Cost dengan GA >= Cost tanpa GA
Input	`SELECT u.name, prod.name, o.total` `FROM users u` `JOIN profiles prof ON u.id = prof.user_id` `JOIN orders o ON u.id = o.user_id` `JOIN products prod ON o.product_id = prod.id WHERE u.email LIKE '%@gmail.com'` `AND prod.category = 'Electronics'` `AND o.total > 100000` `AND prof.bio IS NOT NULL`
Expected Output	- Query Tree di parse dengan benar - Query Tree hasil optimized ekuivalen dengan original tree

	- Query Tree selalu valid dalam proses pipeline
Results	SUCCESS

b. Demo Parse Query

Description	Menguji Parse Query lebih jauh untuk kondisi kondisi yang memungkinkan
Goal	Memastikan sql yang valid akan diparse menjadi query tree yang valid
Method	Parse_Query
Success Criterion	Hasil query tree valid dan ekuivalen dengan sqlnya
Input	`examples = [` `"SELECT id, name FROM users",` `"SELECT * FROM orders WHERE amount > 1000",` `"SELECT a.id, b.name FROM a JOIN b ON a.id = b.a_id WHERE a.x > 5",` `"SELECT u.name, o.total FROM users u JOIN orders o ON u.id = o.user_id WHERE u.age > 25",` `"SELECT * FROM employees WHERE salary BETWEEN 50000 AND 100000",` `"SELECT u.name FROM users u WHERE u.status IN ('active', 'premium', 'trial')",` `"CREATE TABLE orders (id INTEGER PRIMARY KEY, user_id INTEGER FOREIGN KEY REFERENCES users(id), total FLOAT, status VARCHAR(50));",` `"SELECT u.name, o.total, p.name FROM users u JOIN orders o ON u.id = o.user_id JOIN products p ON o.product_id = p.id WHERE u.city = 'Jakarta' AND o.total > 500000"` `]`
Expected Output	 … … …

```
PROJECT
    |-- COLUMN_REF
    |   |-- COLUMN_NAME
    |   |   `-- IDENTIFIER("name")
    |   `-- TABLE_NAME
    |       `-- IDENTIFIER("u")
    |-- COLUMN_REF
    |   |-- COLUMN_NAME
    |   |   `-- IDENTIFIER("total")
    |   `-- TABLE_NAME
    |       `-- IDENTIFIER("o")
    |-- COLUMN_REF
    |   |-- COLUMN_NAME
    |   |   `-- IDENTIFIER("name")
    |   `-- TABLE_NAME
    |       `-- IDENTIFIER("p")
    `-- FILTER
        |-- JOIN("INNER")
        |   |-- JOIN("INNER")
        |   |   |-- ALIAS("u")
        |   |   |   `-- RELATION("users")
        |   |   |-- ALIAS("o")
        |   |   |   `-- RELATION("orders")
        |   |   `-- COMPARISON("=")
        |   |       |-- COLUMN_REF
        |   |       |   |-- COLUMN_NAME
        |   |       |   |   `-- IDENTIFIER("id")
        |   |       |   `-- TABLE_NAME
        |   |       |       `-- IDENTIFIER("u")
        |   |       `-- COLUMN_REF
        |   |           |-- COLUMN_NAME
        |   |           |   `-- IDENTIFIER("user_id")
        |   |           `-- TABLE_NAME
        |   |               `-- IDENTIFIER("o")
```

<table>
<tr><td></td><td> 
</td></tr>
<tr><td>Results</td><td>SUCCESS</td></tr>
</table>

3. Concurrency Control Manager

a. [LockBased] Test grant key to Transaction

Description	Mekanisme pemberian write lock pada satu transaksi terhadap satu objek row
Goal	Memastikan lock Manager memeberikan izin/kunci ketika transaksi meminta akses write pada objek yang belum terkunci
Method	CCManager diinisialisasi dengan algoritma lockbased, sebuah transaksi dimulai, memanggil fungsi validate_object dengan parameter objek tersebut dan ActionType.WRITE. Kemudian dilakukan pengecekan response dari CCM.
Success Criterion	Response dengan atribut allowed bernilai True
Input	`def test_simple_write_lock(self):` `  """Test acquiring write lock"""` `  tid = self.cc_manager.begin_transaction()` `  row = Row(object_id="X", table_name="test_table", data={"value": 100})` `  response = self.cc_manager.validate_object(row, tid, ActionType.WRITE)` `  self.assertTrue(response.allowed)` `  self.cc_manager.end_transaction(tid)`
Expected Output	`Response(allowed=True)`
Result	`test_simple_write_lock (__main__.TestLockBasedAlgorithm.test_simple_write_lock)` `Test acquiring write lock ... COMMITTING Transaction 1: Releasing locks.` `  Released 1 lock(s): ['X']` `ok`

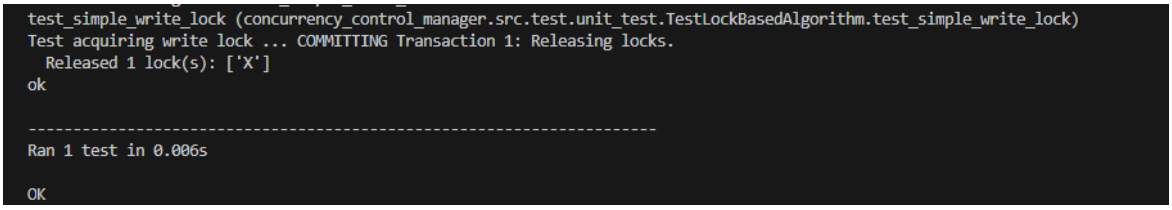
b. [LockBased] Deadlock Prevention (wait-die schema)

Description	Simulasi dua transaksi oleh 2 client untuk mengeksekusi transaksi secara paralel dan skema penanganan deadlock karena saling menunggu kunci akses (cross-dependency)
Goal	Memverifikasi bahwa algoritma concurrency control dapat menangani konflik write-write antara 2 transaksi tanpa menyebabkan deadlock dengan mekanisme wait-die
Method	Inisialisasi QueryProcessor dengan algoritma lockbased, membuat 2 thread untuk simulasi 2 transaksi, membuat skenario cross-dependency, mengamati penanganan oleh CCM yang terjadi
Success Criterion	Transaksi mengikuti skema Wait-Die, transaksi yang lebih muda di-abort dan transaksi yang lebih tua berhasil commit.

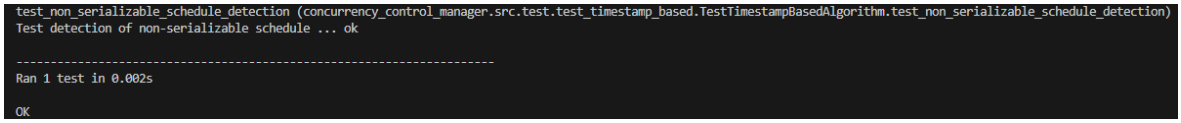
Input	`def run_test_with_algorithm(algorithm_type: AlgorithmType):` `  processor = QueryProcessor()` `  processor.adapter_ccm.set_algorithm(algorithm=algorithm_type)` `  scenario_deadlock_1 = [` `  "BEGIN TRANSACTION",` `  "UPDATE students SET age = 30 WHERE id = 1",` `  "UPDATE courses SET credits = 4 WHERE id = 1",` `  "COMMIT"` `  ]` `  scenario_deadlock_2 = [` `  "BEGIN TRANSACTION",` `  "UPDATE courses SET credits = 3 WHERE id = 1",` `  "UPDATE students SET age = 35 WHERE id = 1",` `  "COMMIT"` `  ]` `  client_a = ClientThread(1, processor, scenario_deadlock_1)` `  client_b = ClientThread(2, processor, scenario_deadlock_2)` `  client_a.start()` `  time.sleep(0.5)  # biar client_a dapat kunci dulu` `  client_b.start()` `  client_a.join()` `  client_b.join()`
Expected Output	Client b yang merupakan transaksi lebih muda di-abort, client a berhasil commit.
Result	`[ABORT] Transaction 2 aborted by CCM: Transaction 2 (Younger) died (aborted) to prevent deadlock.` `[C2] Status: ERROR - Transaction 2 aborted: Transaction 2 (Younger) died (aborted) to prevent deadlock.` `[C2] Transaksi Dibatalkan (Rollback) oleh Sistem.` `--- Client 2 Selesai ---` `found 0 matching rows dari tabel 'courses' (full scan)` `[C1] Status: OK - Query executed successfully.` `[C1] Mengirim: COMMIT`

c. [TimestampBased] Timestamp Ordering Validation

Description	Menguji validasi timestamp ordering pada algoritma Timestamp-Based ketika transaksi dengan timestamp lebih lama mencoba menulis setelah transaksi dengan timestamp lebih baru sudah menulis ke objek yang sama.
Goal	Memastikan algoritma menolak operasi WRITE dari transaksi lama jika transaksi yang lebih baru sudah melakukan WRITE pada objek tersebut (TS(T) < W-TS(X)
Method	1. Buat dua transaksi T1 dan T2, dimana T1.timestamp < T2.timestamp 2. T2 melakukan WRITE pada objek X (harus diizinkan) 3. T1 mencoba melakukan WRITE pada objek X (harus ditolak)
Success Criterion	T2 WRITE: response.allowed = True- T1 WRITE: response.allowed = False
Input	T1 (timestamp: t), T2 (timestamp: t+1), objek X

Expected Output	T2 WRITE diizinkan, T1 WRITE ditolak
Results	

d. [TimestampBased] Non-Serializable Schedule Detection

Description	Menguji kemampuan algoritma dalam mendeteksi dan mencegah schedule yang tidak serializable.
Goal	Memastikan algoritma dapat mendeteksi konflik ketika urutan operasi akan menghasilkan schedule non-serializable dan menolak operasi yang menyebabkan konflik.
Method	1. T1 melakukan READ pada objek A 2. T2 melakukan WRITE pada objek A (diizinkan karena T2 lebih baru) 3. T1 mencoba melakukan WRITE pada objek A (harus ditolak karena T1 < W-TS(A))
Success Criterion	T1 READ: allowed = True- T2 WRITE: allowed = True- T1 WRITE: allowed = False
Input	T1 (timestamp: t), T2 (timestamp: t+1), objek A
Expected Output	T1 READ dan T2 WRITE diizinkan, T1 WRITE ditolak
Results	

e. [ValidationBased] Test Transaction in Execution Always Allowed

Description	Simulasi dua transaksi melakukan WRITE pada objek yang sama.
Goal	Memastikan algoritma menjaga sisi optimistik dimana pengecekan hanya terjadi ketika suatu transaksi melakukan *commit* dan memperbolehkan aksi apapun pada saat eksekusi.
Method	1. T1 melakukan WRITE pada object X 2. T2 melakukan WRITE pada object X 3. Pengecekan apakah status kedua transaksi = allowed 4. T1 commit

	5. T2 commit
Success Criterion	T1 WRITE: response.allowed = True T2 WRITE: response.allowed = True
Input	`def test_optimistic_execution(self):` `    """Test optimistic execution allows operations"""` `    tid1 = self.cc_manager.begin_transaction()` `    tid2 = self.cc_manager.begin_transaction()` `    row = Row(object_id="X", table_name="test_table", data={"value": 100})` `    # Both should be allowed during execution` `    response1 = self.cc_manager.validate_object(row, tid1, ActionType.WRITE)` `    response2 = self.cc_manager.validate_object(row, tid2, ActionType.WRITE)` `    self.assertTrue(response1.allowed)` `    self.assertTrue(response2.allowed)` `    # Validation happens at commit` `    # Clean up` `    try:` `        self.cc_manager.end_transaction(tid1)` `    except:` `        pass` `    try:` `        self.cc_manager.end_transaction(tid2)` `    except:` `        pass`
Expected Output	T1 dan T2 diperbolehkan untuk menulis object X pada saat eksekusi. T1 berhasil *commit*, T2 abort karena konflik write-write dengan T1. Namun, hasil apakah transaksi *commit* atau tidak, tidak dicek pada test ini. Note: Pengecekan status transaksi terdapat pada fungsi lain, contohnya test_write_write_conflict() pada test_validation_based.py
Results	`test_optimistic_execution (__main__.TestValidationBasedAlgorithm.test_optimistic_execution)` `Test optimistic execution allows operations ... ok`

f. [ValidationBased] Test Multiple Read in One Object

Description	Simulasi lebih dari satu klien melakukan READ pada objek yang sama.
Goal	Memastikan algoritma mengubah status semua transaksi menjadi committed karena hanya aksi READ yang terjadi.
Method	1. T1 melakukan READ ke objek Z 2. T2 melakukan READ ke objek Z 3. T3 melakukan READ ke objek Z

	4. T1, T2, dan T3 melakukan *commit* sesuai urutan 5. Pengecekan status transaksi (status = *committed*)
Success Criterion	T1 status = committed T2 status = committed T3 status = committed
Input	`def test_multiple_reads_no_conflict(self):` `"""Test multiple transactions reading the same object"""` `tid1 = self.cc_manager.begin_transaction()` `tid2 = self.cc_manager.begin_transaction()` `tid3 = self.cc_manager.begin_transaction()` `rowZ = Row(object_id="Z", table_name="test_table", data={"value": 500})` `# All read Z` `self.cc_manager.validate_object(rowZ, tid1, ActionType.READ)` `self.cc_manager.validate_object(rowZ, tid2, ActionType.READ)` `self.cc_manager.validate_object(rowZ, tid3, ActionType.READ)` `# All commit` `self.cc_manager.end_transaction(tid1)` `self.cc_manager.end_transaction(tid2)` `self.cc_manager.end_transaction(tid3)` `# All should succeed` `for tid in [tid1, tid2, tid3]:` `status = self.cc_manager.get_transaction_status(tid)` `self.assertIn(status, [TransactionStatus.Committed, TransactionStatus.Terminated])`
Expected Output	T1, T2, dan T3 berhasil melakukan *commit* dan statusnya berubah menjadi *committed*
Results	`........` `----------------------------------------------------------------------` `Ran 8 tests in 0.063s` `OK`

g. [MVCC] Multi-Version Timestamp Ordering (Cascading Rollback dengan Auto Re-execution)

Description	Menguji kemampuan algoritma MVTO dalam menangani cascading rollback. Ketika transaksi menulis ke objek yang sebelumnya dibaca oleh transaksi yang lebih lama, maka transaksi-transaksi lama tersebut harus dibatalkan dan secara otomatis dieksekusi ulang dengan timestamp yang diperbarui.
Goal	Memverifikasi bahwa MVTO dengan benar mengimplementasikan

	mekanisme cascading rollback dan eksekusi ulang otomatis transaksi ketika terjadi konflik timestamp.
Method	1. Membuat 7 transaksi (T1-T7). 2. Mengeksekusi operasi baca: T7 membaca A, T3 membaca B, T1 membaca D. 3. Mengeksekusi operasi tulis: T5 menulis C (nilai=50). 4. Mengeksekusi operasi tulis yang berkonflik: T6 menulis A (nilai=100) - memicu rollback T7 (yang membaca A). 5. Memverifikasi T6 secara otomatis dieksekusi ulang dengan timestamp baru. 6. Memvalidasi bahwa semua operasi selesai dengan sukses setelah eksekusi ulang.
Success Criterion	- Semua operasi dieksekusi tanpa kegagalan permanen. - Cascading rollback ditangani dengan benar. - Transaksi yang dibatalkan secara otomatis dieksekusi ulang. - Execution trace berisi lebih dari 0 operasi yang tercatat. - Test berhasil tanpa assertion error.
Input	- 7 transaksi konkuren - Operasi baca: T7→A, T3→B, T1→D - Operasi tulis: T5→C(50), T6→A(100) - Operasi dieksekusi dalam urutan spesifik untuk memicu konflik timestamp `def test_mvto_case1_cascading_rollback(self` `for i in range(1, 8):` `self.tester.create_transaction(i)` `self.tester.read(7, 'A')` `self.tester.read(3, 'B')` `self.tester.write(5, 'C', 50)` `self.tester.read(1, 'D')` `self.tester.write(6, 'A', 100)` `self.assertGreater(len(self.tester.execution_trace), 0)`
Expected Output	- Operasi tulis T6 ke A menyebabkan T7 (transaksi lama yang membaca A) dibatalkan. - T6 secara otomatis dieksekusi ulang dengan timestamp yang diperbarui. - Semua operasi akhirnya berhasil. - List execution_trace terisi dengan catatan operasi. - assertGreater(len(execution_trace), 0) berhasil.
Results	`test_mvto_case1_cascading_rollback (__main__.TestMVTO.test_mvto_case1_cascading_rollback)` `MVTO Test Case 1: Cascading Rollback with Auto Re-execution ... ok`

h. [MVCC] Multi-Version Two-Phase Locking (Konflik Lock dengan Manajemen Antrian)

Description	Menguji mekanisme resolusi konflik lock MV2PL dengan beberapa transaksi yang mencoba mengakses objek data yang sama. Algoritma menggunakan shared lock untuk operasi baca dan exclusive lock untuk operasi tulis, dengan sistem antrian untuk mengelola transaksi yang menunggu.
Goal	Memverifikasi bahwa MV2PL dengan benar mengelola konflik lock menggunakan pendekatan berbasis antrian, memungkinkan banyak pembaca konkuren sambil memblokir penulis ketika shared lock sedang dipegang.
Method	1. Membuat 5 transaksi (T1-T5) 2. T1, T2, T3 mengakuisisi shared lock dengan membaca objek X secara konkuren 3. T4 mencoba mengakuisisi exclusive lock dengan menulis ke X 4. Memverifikasi T4 diblokir dan ditambahkan ke antrian 5. Memeriksa bahwa operasi baca berhasil dan operasi tulis dengan benar diantrekan 6. Memvalidasi respons mekanisme lock tidak None
Success Criterion	- Beberapa transaksi dapat membaca objek yang sama secara konkuren (shared lock). - Permintaan tulis diblokir ketika shared lock aktif. - Objek respons lock dikembalikan dengan benar. - Manajemen antrian berfungsi dengan benar. - assertIsNotNone(response) berhasil.
Input	- 5 transaksi konkuren. - Operasi baca: T1→X, T2→X, T3→X (shared lock). - Operasi tulis: T4→X(100) (permintaan exclusive lock). - Operasi dirancang untuk menciptakan lock contention. `def test_mv2pl_case1_lock_conflicts(self):` `  for i in range(1, 6):` `  self.tester.create_transaction(i)` `  self.tester.read(1, 'X')` `  self.tester.read(2, 'X')` `  self.tester.read(3, 'X')` `  response = self.tester.write(4, 'X', 100)` `  self.assertIsNotNone(response)`
Expected Output	- T1, T2, T3 berhasil mengakuisisi shared lock untuk membaca X. - Permintaan tulis T4 diblokir karena shared lock yang sudah ada. - Objek respons berisi informasi status lock.

	- execution_trace mencatat semua operasi termasuk status yang diblokir. - Antrian lock dengan benar mengelola transaksi yang menunggu.
Results	`test_mv2pl_case1_lock_conflicts (__main__.TestMV2PL.test_mv2pl_case1_lock_conflicts)` `MV2PL Test Case 1: Lock Conflicts with Queue Management ... ok`

i. [MVCC] Snapshot Isolation dengan First-Updater-Wins (SI-FUW)

Description	Menguji kebijakan resolusi konflik First-Updater-Wins dalam Snapshot Isolation. Ketika beberapa transaksi mencoba memperbarui item data yang sama, transaksi pertama yang mengeluarkan operasi tulis menang, dan transaksi berikutnya yang mencoba menulis item yang sama langsung dibatalkan.
Goal	Memverifikasi bahwa SI-FUW dengan benar mengimplementasikan aturan first-updater-wins dengan mengizinkan operasi tulis pertama dan langsung membatalkan operasi tulis konkuren yang berkonflik.
Method	1. Membuat 5 transaksi (T1-T5). 2. T1 dan T2 keduanya membaca objek X (mendapat snapshot yang sama). 3. T1 menulis ke X terlebih dahulu (nilai=10) - mengakuisisi exclusive lock. 4. T2 mencoba menulis ke X (nilai=20) - harus langsung dibatalkan. 5. Memverifikasi operasi tulis T1 diizinkan. 6. Memverifikasi operasi tulis T2 ditolak (konflik exclusive lock).
Success Criterion	- Operasi tulis pertama (T1) berhasil dan mengakuisisi lock. - Operasi tulis kedua (T2) langsung dibatalkan. - assertTrue(response1.allowed) berhasil. - assertFalse(response2.allowed) berhasil. - Deteksi konflik terjadi pada waktu tulis, bukan waktu commit.
Input	- 5 transaksi konkuren. - Operasi baca: T1→X, T2→X (keduanya mendapat snapshot yang sama). - Operasi tulis: T1→X(10), T2→X(20). - Kedua operasi tulis menargetkan objek yang sama untuk memicu konflik. `def test_snapshot_fuw_case1_basic_fuw(self):` `  for i in range(1, 6):` `  self.tester.create_transaction(i)` `  self.tester.read(1, 'X')` `  self.tester.read(2, 'X')` `  response1 = self.tester.write(1, 'X', 10)` `  response2 = self.tester.write(2, 'X', 20)` `  self.assertTrue(response1.allowed)`

	`self.assertFalse(response2.allowed)`
Expected Output	- Operasi tulis T1 berhasil: response1.allowed = True. - Operasi tulis T2 gagal: response2.allowed = False. - T2 menerima pesan "exclusive lock conflict". - execution_trace menunjukkan T1 berhasil dan T2 dibatalkan. - Updater pertama memenangkan konflik.
Results	`SI-FUW Test Case 1: Basic First-Updater-Wins ... ok` `test_snapshot_fuw_case2_multiple_conflicts (__main__.TestSnapshotFUW.test_snapshot_fuw_case2_multiple_conflicts)`

j. [MVCC] Multi-Version Two-Phase Locking (Konflik Lock dengan Manajemen Antrian)

Description	Menguji kebijakan resolusi konflik First-Committer-Wins dalam Snapshot Isolation. Berbeda dengan FUW, FCW mengizinkan operasi tulis konkuren untuk di-buffer dan hanya mendeteksi konflik pada waktu commit. Transaksi pertama yang commit menang, dan transaksi berikutnya yang mencoba commit dengan operasi tulis yang berkonflik akan dibatalkan.
Goal	Memverifikasi bahwa SI-FCW dengan benar mengimplementasikan deteksi konflik yang ditunda dengan mengizinkan operasi tulis konkuren selama eksekusi dan menyelesaikan konflik hanya pada waktu commit berdasarkan urutan commit.
Method	1. Membuat 5 transaksi (T1-T5). 2. T1 dan T2 keduanya membaca objek X (mendapat snapshot yang sama). 3. T1 menulis ke X (nilai=10) - di-buffer, belum di-commit. 4. T2 menulis ke X (nilai=20) - juga di-buffer. 5. T1 commit terlebih dahulu - harus berhasil. 6. T2 commit kedua - harus dibatalkan (terdeteksi konflik write-write).
Success Criterion	- Kedua operasi tulis berhasil di-buffer selama eksekusi. - Commit pertama (T1) berhasil. - Commit kedua (T2) dibatalkan karena konflik write-write. - assertTrue(result1.allowed) berhasil. - assertFalse(result2.allowed) berhasil. - Deteksi konflik ditunda hingga fase commit.
Input	- 5 transaksi konkuren - Operasi baca: T1→X, T2→X (snapshot yang sama) - Operasi tulis: T1→X(10), T2→X(20) (keduanya di-buffer) - Operasi commit: T1 commit, kemudian T2 commit - Kedua transaksi menulis ke objek X yang sama `def test_snapshot_fcw_case1_basic_fcw(self):`

	`for i in range(1, 6):` `  self.tester.create_transaction(i)` `self.tester.read(1, 'X')` `self.tester.read(2, 'X')` `self.tester.write(1, 'X', 10)` `self.tester.write(2, 'X', 20)` `result1 = self.tester.commit(1)` `result2 = self.tester.commit(2)` `self.assertTrue(result1.allowed)` `self.assertFalse(result2.allowed)`
Expected Output	- Kedua operasi tulis diterima selama fase eksekusi. - T1 commit berhasil: result1.allowed = True. - T2 commit gagal: result2.allowed = False. - T2 menerima pesan error "write-write conflict". - execution_trace menunjukkan kedua tulis di-buffer, T1 commit OK, T2 commit FAIL.
Results	`SI-FCW Test Case 1: Basic First-Committer-Wins ... ok` `test_snapshot_fcw_case2_multiple_items (__main__.TestSnapshotFCW.test_snapshot_fcw_case2_multiple_items)`

4. Storage Manager

a. Create Table

Description	Menguji fungsi create_table() untuk memastikan tabel baru dapat dibuat dan validasi error berjalan dengan benar.
Goal	Memastikan Storage Manager dapat membuat tabel baru dengan nama valid dan mendeteksi error seperti nama duplikat atau nama tidak valid.
Method	Menjalankan create_table() dengan beberapa skenario: valid create, duplicate create, invalid name.
Success Criterion	Tabel muncul dalam daftar self.sm.tables ketika berhasil dibuat; exception ValueError muncul untuk kondisi invalid.
Input	`- create_table("users", ["id", "name", "email"])\` `- create_table("users", ["id"])` `- create_table("123invalid", ["id"]) |`
Expected Output	`- Tabel "users" berhasil dibuat` `- Duplicate creation melempar ValueError` `- Invalid name melempar ValueError`
Result	SUCCESS

b. Read Block With Filtering

Description	Menguji fungsi read_block() untuk mengambil data dengan filter dan projection.
Goal	Memastikan Storage Manager dapat memfilter row berdasarkan kondisi (WHERE) dan memproyeksikan kolom tertentu.
Method	Insert 8 rows ke tabel mahasiswa, lalu uji: no filter, projection, dan multi-condition filter.
Success Criterion	Jumlah row sesuai dengan kondisi filter, dan kolom yang dikembalikan sesuai dengan projection.
Input	- 8 rows mahasiswa\ - Query 1: tanpa filter - Query 2: projection kolom nim,nama,ipk - Query 3: WHERE ipk >= 3.5 - Query 4: WHERE ipk >= 3.5 AND angkatan = 2021
Expected Output	- Query 1 → 8 rows - Query 2 → 8 rows, tiap row hanya 3 kolom - Query 3 → 5 rows - Query 4 → 2 rows
Result	SUCCESS

5. Failure Recovery

A. Test Log Buffering

Field	Content
Description	Menguji mekanisme *buffering* log di memori sebelum ditulis ke disk.
Goal	Memastikan efisiensi I/O dengan menampung log operasi tulis (WRITE) di memori (mem_wal) terlebih dahulu dan tidak langsung menulisnya ke penyimpanan fisik (virtual_disk).

Method	Melakukan satu operasi penulisan log (write_log) dan memeriksa jumlah entri di mem_wal serta virtual_disk.
Success Criterion	mem_wal berisi 1 entri, sedangkan virtual_disk masih kosong (0).
Input	`# SPESIFIKASI 1: BUFFERING` `def test_write_log_memory_buffering(self):` `    self.frm.write_log(ExecutionResult(1, actiontype.write, datetime.now()))` `    self.assertEqual(len(self.frm.mem_wal), 1)` `    self.assertEqual(len(self.virtual_disk), 0)`
Expected Output	Log tersimpan di buffer memori, disk tetap kosong.
Result	Success

B. Test Flush on Commit

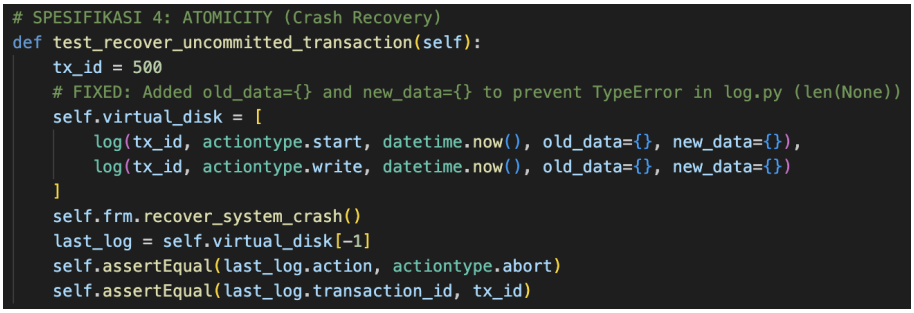
Field	Content
Description	Menguji mekanisme *flushing* (penulisan paksa) log dari memori ke disk saat terjadi transaksi *commit*.
Goal	Memastikan prinsip *Durability* dengan menjamin semua log transaksi yang di-*commit* tersimpan permanen di disk.
Method	Menulis log operasi WRITE diikuti dengan log COMMIT. Memeriksa apakah virtual_disk telah terisi.
Success Criterion	Jumlah log di virtual_disk sesuai dengan jumlah operasi yang dilakukan (2 entri), menandakan data telah dipindahkan dari memori sementara ke penyimpanan.
Input	`# SPESIFIKASI 2: FLUSH ON COMMIT` `def test_write_log_flush_on_commit(self):` `    self.frm.write_log(ExecutionResult(1, actiontype.write, datetime.now()))` `    self.frm.write_log(ExecutionResult(1, actiontype.commit, datetime.now()))` `    # FIXED: The implementation of write_log flushes to disk but DOES NOT clear mem_wal.` `    # So mem_wal will still have 2 items.` `    self.assertEqual(len(self.frm.mem_wal), 2)` `    self.assertEqual(len(self.virtual_disk), 2)`
Expected Output	virtual_disk berisi 2 log.
Result	Success

C. Test Flush on Limit (WAL Size)

Field	Content
Description	Menguji mekanisme otomatis *flush* log ke disk ketika ukuran buffer memori melebihi batas yang ditentukan (wal_size).
Goal	Mencegah kehabisan memori (*Memory Overflow*) dengan memindahkan log ke disk secara berkala.
Method	Mengatur wal_size ke nilai kecil (1), lalu melakukan penulisan log hingga jumlahnya melebihi batas tersebut.
Success Criterion	Ketika jumlah log > wal_size, log otomatis ditulis ke virtual_disk.
Input	`# SPESIFIKASI 3: FLUSH ON LIMIT` `def test_wal_size_limit_trigger_flush(self):` `    # FIXED: Logic is len(mem) > wal_size. If wal_size is 2, we need 3 items to flush.` `    # Setting wal_size to 1 ensures that 2 items > 1 triggers flush.` `    self.frm.wal_size = 1` `    self.frm.write_log(ExecutionResult(1, actiontype.write, datetime.now()))` `    # 1 > 1 is False, no flush yet` `    self.assertEqual(len(self.virtual_disk), 0)` `    self.frm.write_log(ExecutionResult(1, actiontype.write, datetime.now()))` `    # 2 > 1 is True, flush happens` `    self.assertEqual(len(self.virtual_disk), 2)` `    # FIXED: Implementation does not clear mem_wal after limit flush` `    self.assertEqual(len(self.frm.mem_wal), 2)`
Expected Output	virtual_disk terisi 2 log setelah operasi kedua dilakukan.
Result	Success

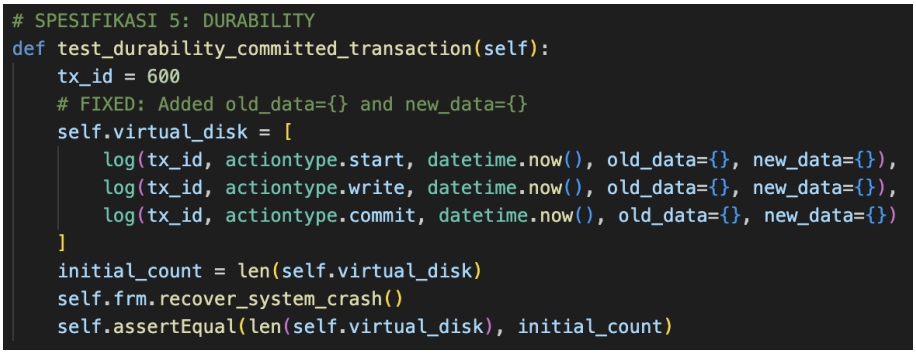
D. Test Crash Recovery (Atomicity)

Field	Content
Description	Menguji mekanisme pemulihan sistem saat terjadi *crash* di tengah transaksi yang belum selesai (*uncommitted*). Skenario ini mensimulasikan kegagalan sistem setelah operasi WRITE tetapi sebelum COMMIT.
Goal	Memastikan prinsip Atomicity terpenuhi, di mana transaksi yang belum *commit* saat *crash* harus dibatalkan (*Aborted*) dan perubahan

	datanya dikembalikan (*Undone*) saat sistem pulih.
Method	Menginisialisasi virtual_disk dengan log transaksi yang tidak lengkap (hanya ada log START dan WRITE). Kemudian memanggil fungsi recover_system_crash() dan memeriksa log terakhir yang ditambahkan oleh sistem.
Success Criterion	Sistem secara otomatis mendeteksi transaksi gantung dan menambahkan log ABORT untuk transaksi tersebut.
Input	
Expected Output	Log terakhir pada disk harus bertipe actiontype.abort dengan transaction_id yang sesuai (500), menandakan sistem membatalkan transaksi tersebut.
Result	Success

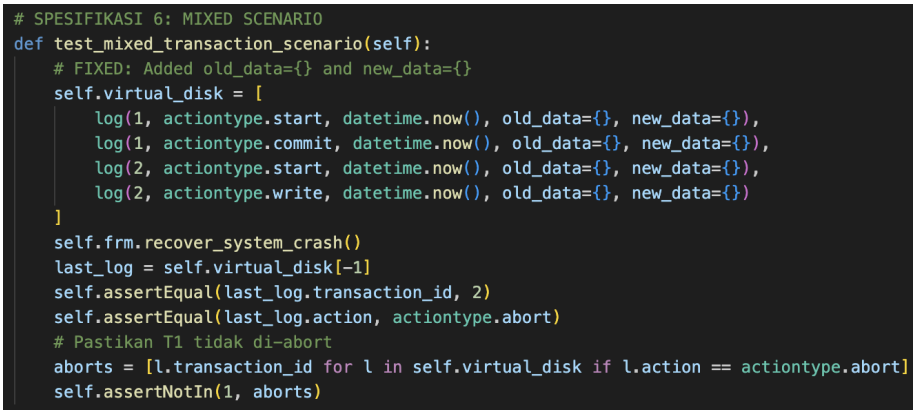
E. Test Durability (Committed Transaction)

Field	Content
Description	Menguji ketahanan data (*Durability*) untuk transaksi yang telah berhasil melakukan COMMIT sebelum terjadi kegagalan sistem.
Goal	Memastikan bahwa transaksi yang sudah *commit* tidak dibatalkan atau diubah saat proses *recovery* berjalan. Data harus tetap persisten.
Method	Menginisialisasi virtual_disk dengan rangkaian log transaksi yang lengkap (START -> WRITE -> COMMIT). Kemudian memanggil fungsi recover_system_crash() dan memverifikasi jumlah log.
Success Criterion	Sistem tidak melakukan operasi ABORT atau UNDO pada transaksi tersebut. Jumlah log dalam disk harus tetap sama (tidak ada penambahan log abort).

Input	
Expected Output	Panjang list virtual_disk setelah recovery sama dengan panjang awal, menunjukkan bahwa transaksi yang sudah *commit* diabaikan oleh proses *undo* dan dianggap aman.
Result	Success

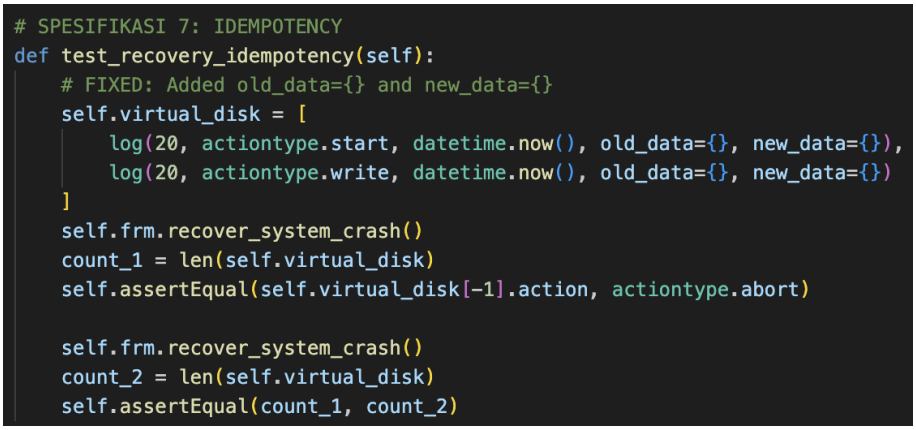
F. Test Mixed Scenario Recovery

Field	Content
Description	Menguji pemulihan sistem dalam kondisi campuran, di mana terdapat transaksi yang sudah selesai (*committed*) dan transaksi yang belum selesai (*uncommitted*) secara bersamaan saat *crash* terjadi.
Goal	Memastikan sistem mampu memilah transaksi: Transaksi *committed* dipertahankan (*Durability*), sedangkan transaksi *uncommitted* dibatalkan (*Atomicity*).
Method	Menyiapkan log simulasi untuk T1 (sudah commit) dan T2 (belum commit). Menjalankan recover_system_crash() dan memeriksa status akhir kedua transaksi.
Success Criterion	T2 diakhiri dengan log ABORT, sedangkan T1 tidak di-*abort* (tetap valid).

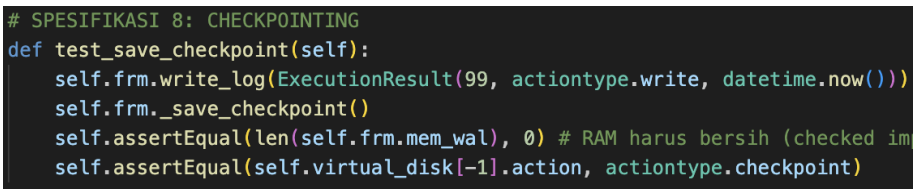
Input	
Expected Output	Transaksi 2 di-*abort*, Transaksi 1 aman.
Result	Success

G. Test Recovery Idempotency

Field	Content
Description	Menguji sifat Idempotency dari proses *recovery*, yaitu jika proses *recovery* dijalankan berulang kali pada *state* yang sama, hasilnya harus tetap konsisten dan tidak merusak data (misalnya tidak melakukan *double-undo*).
Goal	Memastikan menjalankan recover_system_crash() dua kali berturut-turut tidak menambah log ABORT duplikat atau mengubah status sistem yang sudah pulih.
Method	Menjalankan recovery pertama kali pada log transaksi yang belum selesai, mencatat jumlah log. Kemudian menjalankan recovery kedua kalinya dan membandingkan jumlah log akhir.
Success Criterion	Jumlah log setelah recovery kedua sama persis dengan jumlah log setelah recovery pertama.

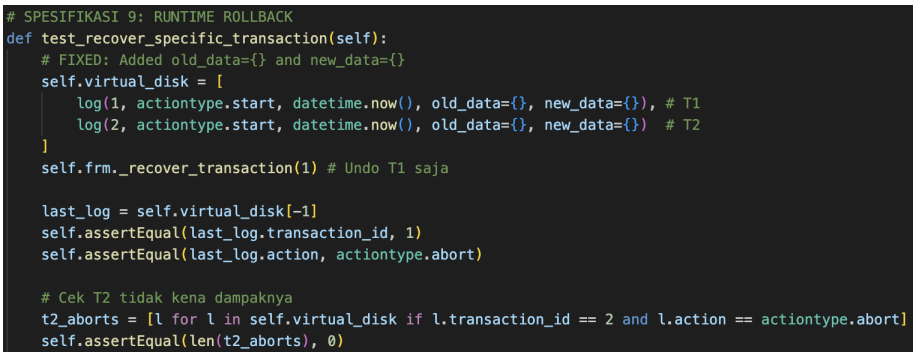
Input	
Expected Output	count_1 sama dengan count_2. Recovery kedua tidak menambahkan entri log baru karena transaksi sudah di-abort pada recovery pertama.
Result	Success

H. Test Checkpointing

Field	Content
Description	Menguji mekanisme pembuatan *checkpoint* manual. Checkpoint digunakan untuk menyinkronkan data di memori ke disk dan mengurangi waktu *recovery* di masa depan.
Goal	Memastikan bahwa saat *checkpoint* dipicu, semua log di memori (mem_wal) dikosongkan (di-*flush*) dan sebuah log bertipe CHECKPOINT ditulis ke disk.
Method	Menulis sebuah log transaksi ke memori FRM, kemudian memanggil fungsi _save_checkpoint(). Setelah itu, memeriksa apakah memori kosong dan apakah log terakhir di disk adalah checkpoint.
Success Criterion	Buffer memori (mem_wal) menjadi kosong (0) dan entri terakhir pada *virtual disk* memiliki tipe aksi checkpoint.
Input	
Expected Output	len(self.frm.mem_wal) bernilai 0 dan self.virtual_disk[-1].action adalah actiontype.checkpoint.

Result	Success

I. Test Runtime Rollback (Transaction Specific)

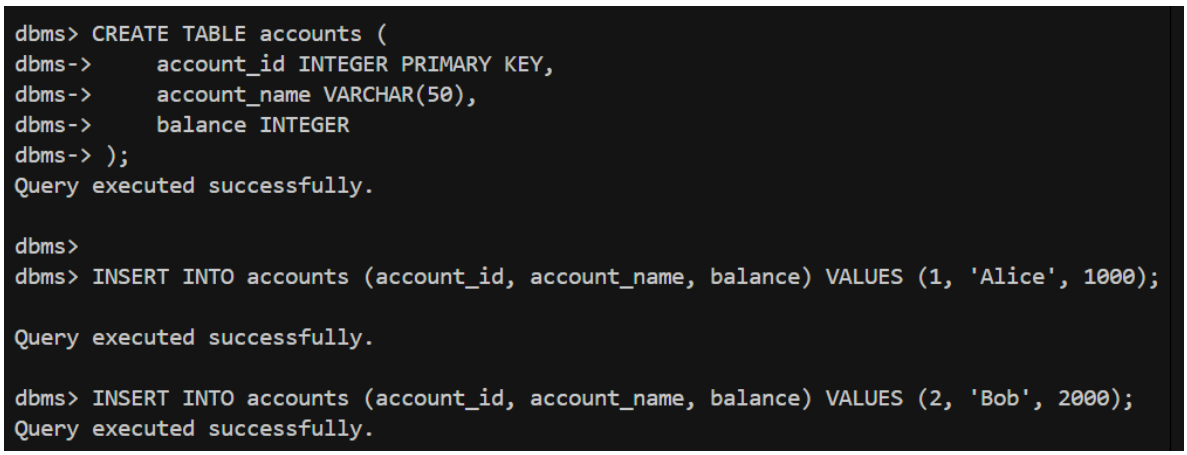
Field	Content
Description	Menguji kemampuan sistem untuk membatalkan (*rollback*) satu transaksi spesifik secara terisolasi saat sistem sedang berjalan (bukan akibat *crash*), misalnya karena permintaan user atau *deadlock*.
Goal	Memastikan hanya transaksi yang ditargetkan yang dibatalkan, sementara transaksi lain yang berjalan bersamaan tidak terpengaruh.
Method	Menyiapkan log simulasi untuk dua transaksi (T1 dan T2) yang sedang berjalan. Memanggil _recover_transaction(1)untuk membatalkan T1 saja, lalu memeriksa log.
Success Criterion	Terdapat log ABORT untuk transaksi T1 di akhir disk, namun tidak ada log ABORT untuk transaksi T2.
Input	
Expected Output	Log terakhir adalah abort untuk T1. List t2_aborts kosong (T2 tidak di-abort).
Result	Success

```
# SPESIFIKASI 9: RUNTIME ROLLBACK
def test_recover_specific_transaction(self):
    # FIXED: Added old_data={} and new_data={}
    self.virtual_disk = [
        log(1, actiontype.start, datetime.now(), old_data={}, new_data={}), # T1
        log(2, actiontype.start, datetime.now(), old_data={}, new_data={})  # T2
    ]
    self.frm._recover_transaction(1) # Undo T1 saja

    last_log = self.virtual_disk[-1]
    self.assertEqual(last_log.transaction_id, 1)
    self.assertEqual(last_log.action, actiontype.abort)

    # Cek T2 tidak kena dampaknya
    t2_aborts = [l for l in self.virtual_disk if l.transaction_id == 2 and l.action == actiontype.abort]
    self.assertEqual(len(t2_aborts), 0)
```

6. Database Management System

Berikan **5 contoh *system testing***, di mana setiap komponen harus terlibat dalam setidaknya satu skema pengujian. Setiap pengujian dapat melibatkan lebih dari satu komponen.

a. Concurrent Transaction and Recovery

Description	Memeriksa fungsionalitas socket untuk mengurus multiple *terminal* dan juga *deadlock prevention* dan *recovery*. Meliputi FRM untuk logging dan recovery, CCM untuk locking, QO untuk parsing query, QP untuk eksekusi *query*, dan SM untuk operasi yang meliputi data.
Goal	Menguji wait-die dan interaksi banyak terminal.
Method	Menjalankan query yang mengarah WRITE ke tabel yang sama.
Success Criterion	Deadlock dihentikan dan integritas data dijaga.
Setup	
Demo	1. Update di Terminal 1

```
dbms> CREATE TABLE accounts (
dbms->     account_id INTEGER PRIMARY KEY,
dbms->     account_name VARCHAR(50),
dbms->     balance INTEGER
dbms-> );
Query executed successfully.

dbms>
dbms> INSERT INTO accounts (account_id, account_name, balance) VALUES (1, 'Alice', 1000);

Query executed successfully.

dbms> INSERT INTO accounts (account_id, account_name, balance) VALUES (2, 'Bob', 2000);
Query executed successfully.
```

```
dbms> BEGIN TRANSACTION;
Transaction started. ID=4

dbms*> SELECT * FROM accounts WHERE account_id = 1;
account_id | account_name | balance
-----------+--------------+--------
1          | Alice        | 1000

(1 row)

dbms*> UPDATE accounts SET balance = 1500 WHERE acc
ount_id = 1;
Query executed successfully.
```

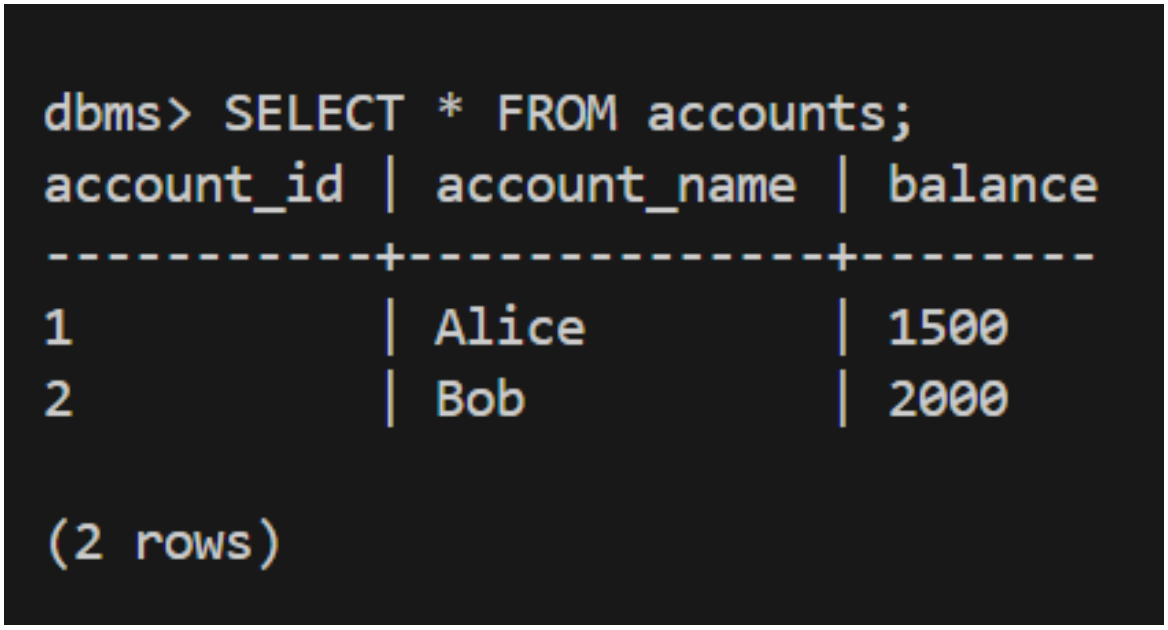
2. Update di Terminal 2 (Killed)

```
dbms> BEGIN TRANSACTION;
Transaction started. ID=5

dbms*> SELECT * FROM accounts WHERE account_id = 1;
ERROR: Transaction 5 aborted: Transaction 5 (Younger) died (aborted) to prevent deadlock.

dbms*> UPDATE accounts SET balance = 1200 WHERE account_id = 1;
ERROR: Transaction 6 aborted: Transaction 6 (Younger) died (aborted) to prevent deadlock.

dbms*>
```

3. Terminal 1 Commit

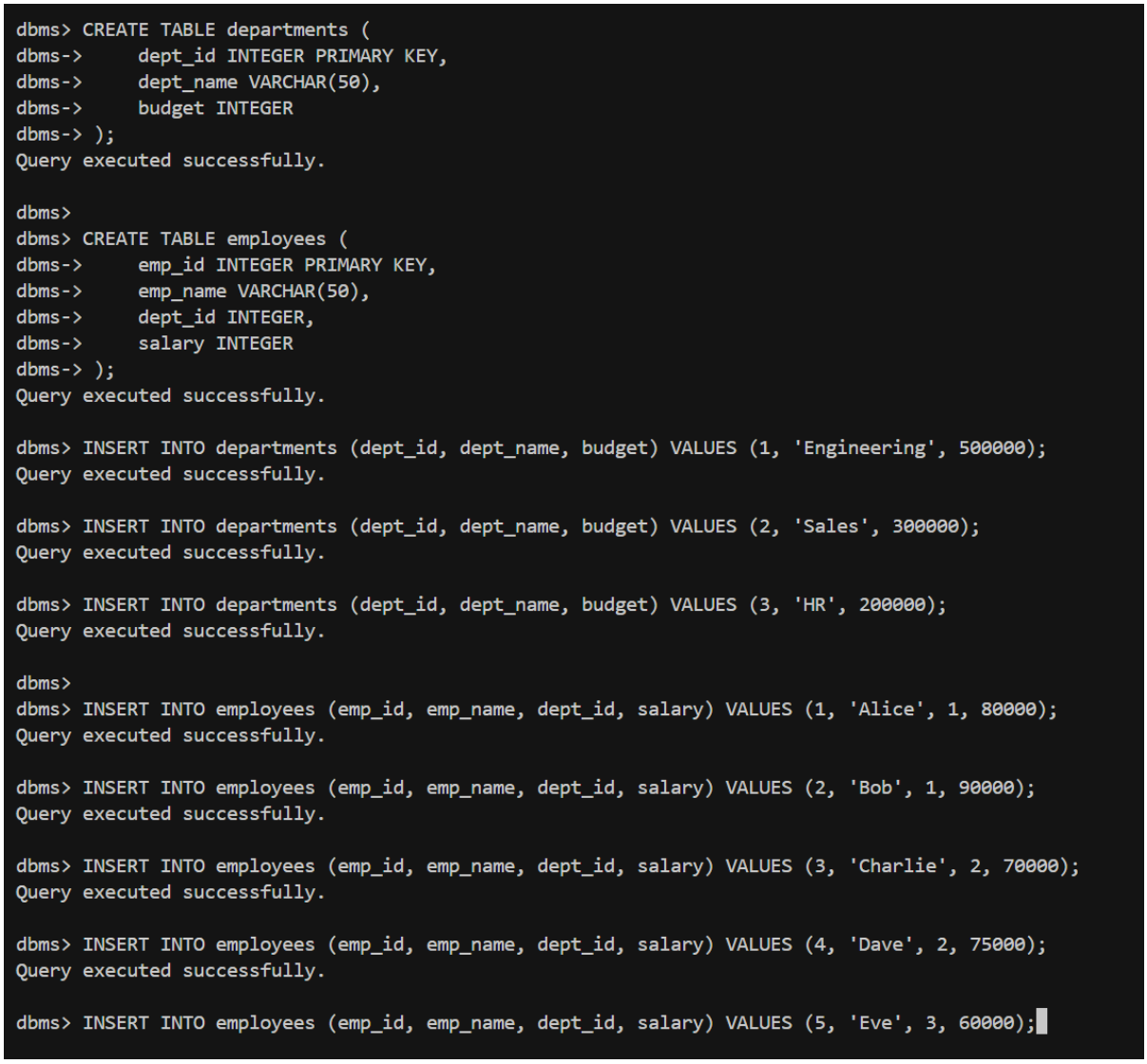
```
dbms*> COMMIT;
Transaction 4 COMMITed successfully.
```

4. Verify

	 5. Aborted transaction does not affect Alice
Result	SUCCESS

b. Complex Query Parsing

Description	Memeriksa fungsionalitas program untuk dapat mengurus berbagai jenis query kompleks yang merupakan kumpulan query-query sederhana. Meliputi QO untuk parsing *query*, QP untuk eksekusi *query*, dan SM untuk operasi yang meliputi data.
Goal	Menguji keberhasilan program membaca dan memproses *query* yang sulit
Method	Mencoba di terminal beberapa *query* SELECT yang kompleks.
Success Criterion	Dapat memberikan hasil.

Setup	
Demo	1. Verify Tables

```
dbms> CREATE TABLE departments (
dbms->     dept_id INTEGER PRIMARY KEY,
dbms->     dept_name VARCHAR(50),
dbms->     budget INTEGER
dbms-> );
Query executed successfully.

dbms>
dbms> CREATE TABLE employees (
dbms->     emp_id INTEGER PRIMARY KEY,
dbms->     emp_name VARCHAR(50),
dbms->     dept_id INTEGER,
dbms->     salary INTEGER
dbms-> );
Query executed successfully.

dbms> INSERT INTO departments (dept_id, dept_name, budget) VALUES (1, 'Engineering', 500000);
Query executed successfully.

dbms> INSERT INTO departments (dept_id, dept_name, budget) VALUES (2, 'Sales', 300000);
Query executed successfully.

dbms> INSERT INTO departments (dept_id, dept_name, budget) VALUES (3, 'HR', 200000);
Query executed successfully.

dbms>
dbms> INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES (1, 'Alice', 1, 80000);
Query executed successfully.

dbms> INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES (2, 'Bob', 1, 90000);
Query executed successfully.

dbms> INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES (3, 'Charlie', 2, 70000);
Query executed successfully.

dbms> INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES (4, 'Dave', 2, 75000);
Query executed successfully.

dbms> INSERT INTO employees (emp_id, emp_name, dept_id, salary) VALUES (5, 'Eve', 3, 60000);
```

```
dbms> SELECT * FROM employees WHERE dept_id = 1;
emp_id | emp_name | dept_id | salary
-------+----------+---------+-------
1      | Alice    | 1       | 80000
2      | Bob      | 1       | 90000

(2 rows)

dbms> SELECT * FROM employees WHERE salary > 70000 ORDER BY salary;
emp_id | emp_name | dept_id | salary
-------+----------+---------+-------
4      | Dave     | 2       | 75000
1      | Alice    | 1       | 80000
2      | Bob      | 1       | 90000

(3 rows)
```

2. Test Join

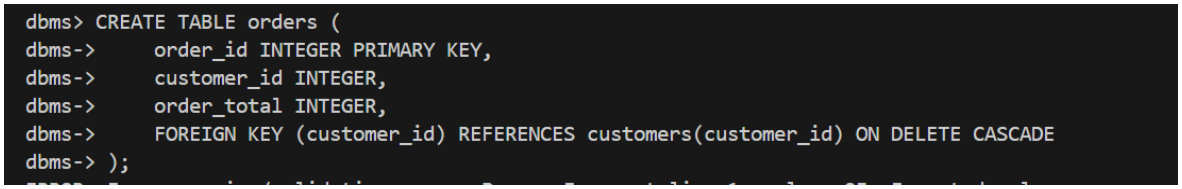
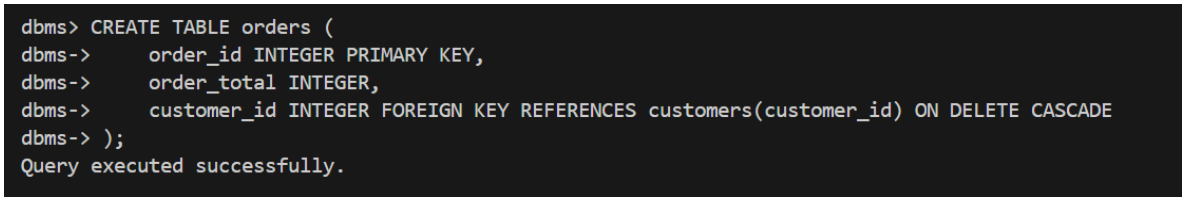
```
dbms> SELECT e.emp_name, d.dept_name, e.salary
dbms-> FROM employees e
dbms-> JOIN departments d ON e.dept_id = d.dept_id
dbms-> WHERE e.salary > 70000;
emp_name | dept_name   | salary
---------+-------------+-------
Alice    | Engineering | 80000
Bob      | Engineering | 90000
Dave     | Engineering | 75000
Alice    | Sales       | 80000
Bob      | Sales       | 90000
Dave     | Sales       | 75000
Alice    | HR          | 80000
Bob      | HR          | 90000
Dave     | HR          | 75000

(9 rows)
```

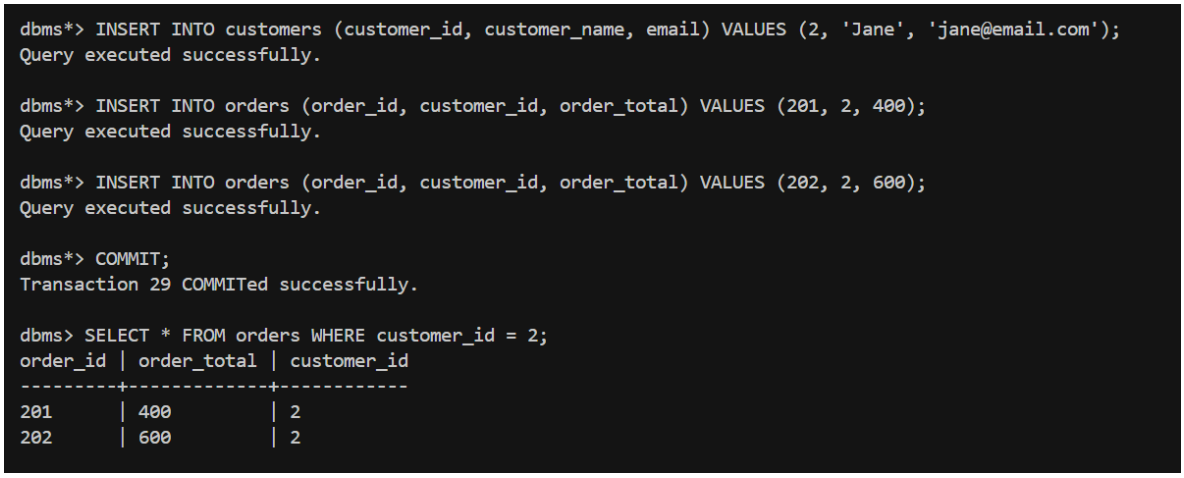
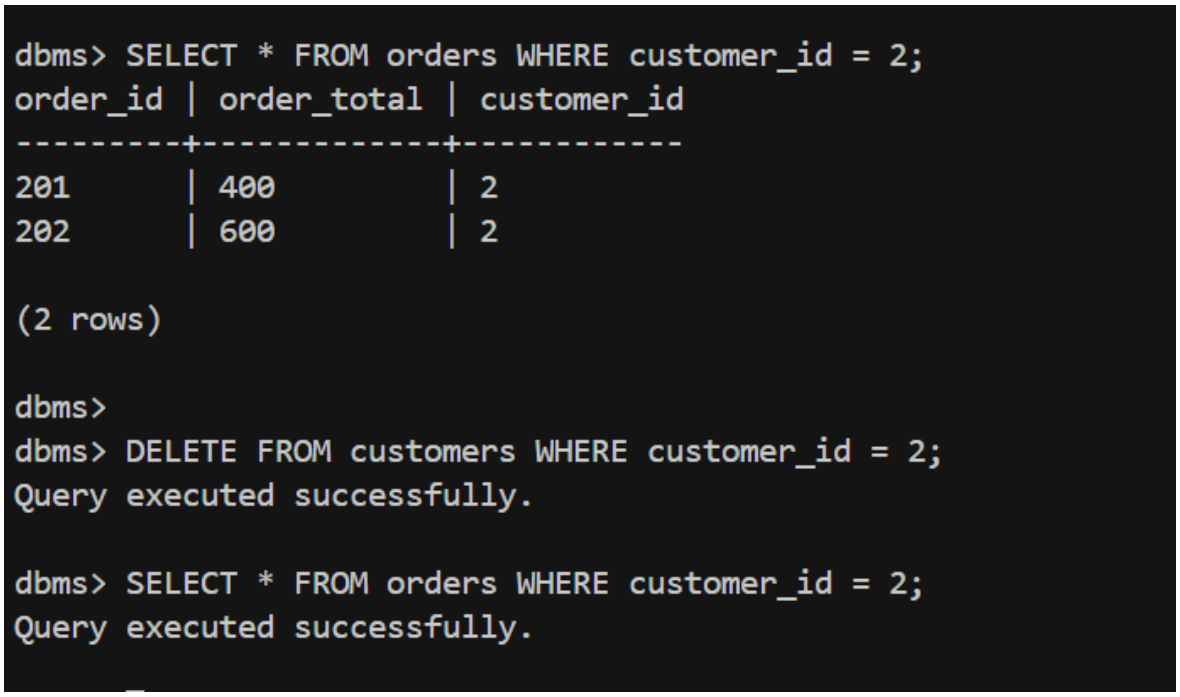
3. Test Complex Query

	`` dbms> SELECT * FROM employees dbms-> WHERE dept_id IN (SELECT dept_id FROM departments WHERE budget > 250000) dbms-> ORDER BY salary DESC dbms-> LIMIT 3; emp_id | emp_name | dept_id | salary -------+---------+---------+------- 2 | Bob | 1 | 90000 1 | Alice | 1 | 80000 4 | Dave | 2 | 75000 (3 rows) ``` 4. Test Alias and Join ``` dbms> SELECT e.emp_name, d.dept_name dbms-> FROM employees e dbms-> JOIN departments d ON e.dept_id = d.dept_id dbms-> WHERE e.salary > 70000 AND d.budget > 300000 dbms-> ORDER BY e.salary DESC; emp_name | dept_name ---------+------------ Alice | Engineering Bob | Engineering Dave | Engineering (3 rows) ```
Result	SUCCESS

c. Multi-Table Key Constraints Transaction

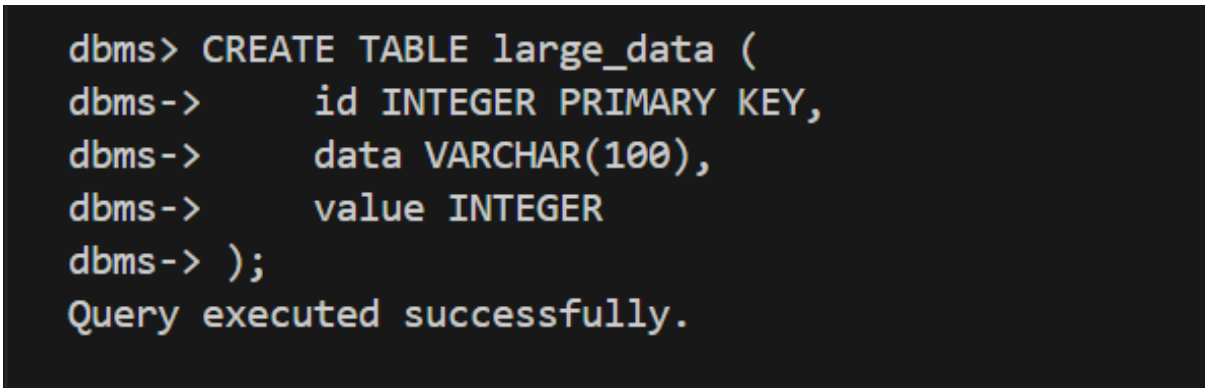
Description	Memeriksa penjagaan integritas berdasarkan parameter CREATE TABLE dalam hal DROP TABLE dan DELETE. Meliputi utamanya Storage Manager dengan Query Processor.
Goal	Menguji constraint CASCADE.

Method	Membuat tabel dengan konstrain, dan delete.
Success Criterion	Query penghapusan di satu tabel mempengaruhi tabel lainnya.
Setup	  

	
Demo	Delete customer_id = 2, verify cascading delete. 
Result	SUCCESS

d. Long Transaction Checkpoint and Recovery Test

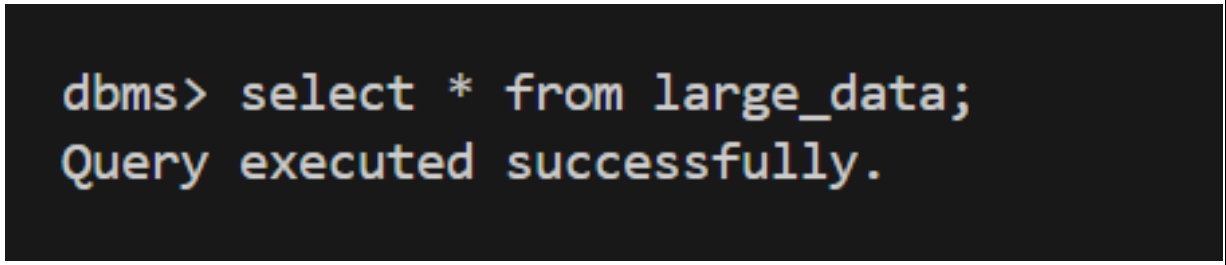
Description	Memeriksa fungsionalitas *checkpoint* dan *system failure recovery*. Test ini meliputi utamanya QP dan FRM.

Goal	Menguji apakah walaupun setelah *checkpoint* (write to storage manager), jika belum commit, akan kembali ke kondisi semula dalam hal system failure.
Method	Mulai *transaction*, dan melakukan banyak operasi hingga memenuhi WAL dan terpaksa menggunakan *checkpoint*. Di tengah *transaction*, kill terminal. Setelah itu, cek apakah operasi yang telah dimasukkan melalui checkpoint tersimpan atau tidak.
Success Criterion	Integritas data dijaga dengan tabel yang kosong.
Setup	
Demo	1. Many inserts and updates (pass through checkpoint)

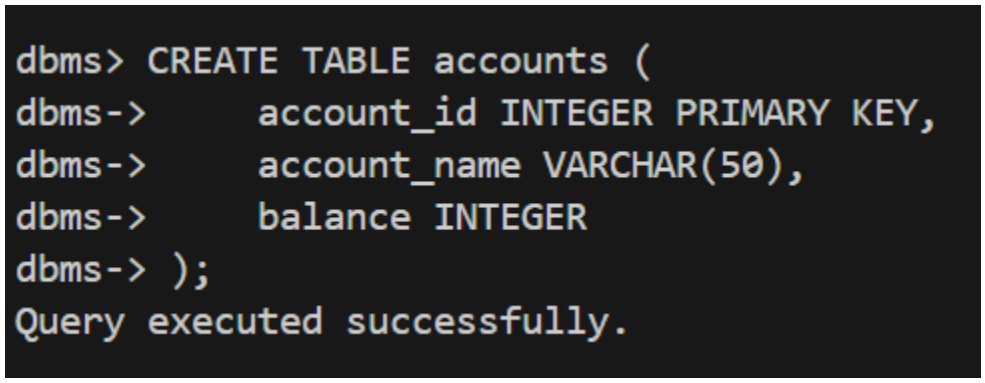
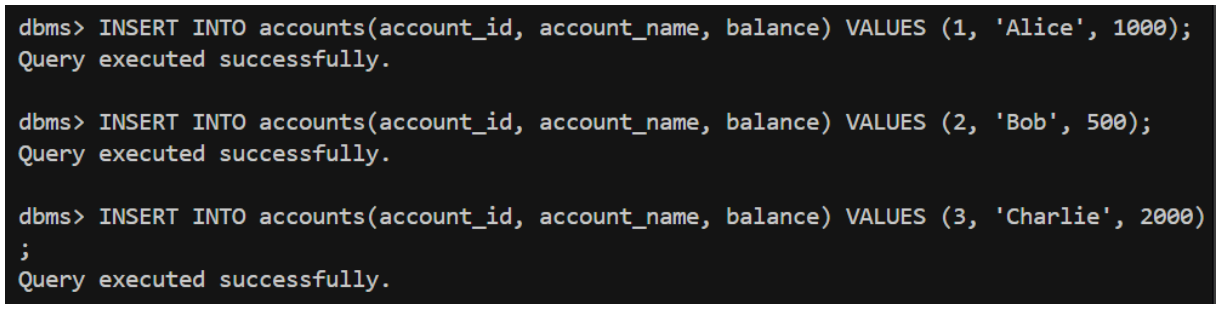
```
dbms> BEGIN TRANSACTION;
Transaction started. ID=34

dbms*> INSERT INTO large_data (id, data, value) VALUES (1, 'data_1', 100);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (2, 'data_2', 200);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (3, 'data_3', 300);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (4, 'data_4', 400);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (5, 'data_5', 500);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (6, 'data_6', 600);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (7, 'data_7', 700);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (8, 'data_8', 800);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (9, 'data_9', 900);
Query executed successfully.

dbms*> INSERT INTO large_data (id, data, value) VALUES (10, 'data_10', 1000);
Query executed successfully.
```

```
dbms*> UPDATE large_data SET value = value + 100 WHERE id > 5;
Query executed successfully.

dbms*>
dbms*> DELETE FROM large_data WHERE id = 1;
Query executed successfully.

dbms*> DELETE FROM large_data WHERE id = 10;
Query executed successfully.
```

2. Kill Server (close terminal)
3. Restart Server and run Client
4. Verify

	
Result	SUCCESS

e. Transaction Isolation Test

Goal	Menguji isolasi transaksi
Method	Melakukan pembacaan data terhadap data yang dirubah
Success Criterion	Tidak membaca hasil perubahan yang dilakukan transaksi lain.
Setup	 

Demo	``` dbms> BEGIN TRANSACTION; Transaction started. ID=5 ``` ``` dbms> BEGIN TRANSACTION; Transaction started. ID=6 ``` (transaction 1) ``` dbms*> UPDATE accounts SET balance = balance - 200 WHE RE account_id = 1; Query executed successfully. dbms*> SELECT * FROM accounts WHERE account_id = 1; account_id | account_name | balance ----------+--------------+-------- 1 | Alice | 800 (1 row) ``` (transaction 2 di saat bersamaan) ``` dbms*> SELECT * FROM accounts WHERE account_i d = 1; ERROR: Transaction 6 aborted: Transaction 6 (Younger) died (aborted) to prevent deadlock. ```
Result	SUCCESS

Part IV. Distribusi Workload

Tuliskan nama-nama anggota yang berkontribusi. Apabila ada anggota yang tidak berkontribusi, tidak perlu dicantumkan.

1. Query Processor

NIM	Nama	Workload
13523066	Muhammad Ghifary Komara P	Laporan, main, adapter_optimizer, desain awal kelas QueryProcessor, utility functions, client testing
13523114	Guntara Hambali	Laporan, threading, adapter_ccm, client class, testing, debugging transaction
13523084	Lutfi Hakim Yusra	Laporan, query_execution, TransactionBuffer, buffer to storage, adapter_frm, handle recover_transaction, handle recover_system_failure, execute_query, execute_node.
13523028	Muhammad Aditya Rahmadeni	Laporan, adapter_storage, testing

2. Query Optimizer

NIM	Nama	Workload
13523068	Muh. Rusmin Nurwadin	Laporan, parser, Rule 4, 6, 7
13523088	Aryo Bama Wiratama	Laporan, tokenizer, get_cost, Rule 3

13523076	Nadhif Al Rozin	Laporan, check_query, GA, Rule 1, 2, 8, Readme
13523070	Sebastian Hung Yansen	Laporan, query_tree, Rule 5

3. Concurrency Control Manager

NIM	Nama	Workload
13522004	Eduardus Alvito Kristiadi	TimestampBased Algorithm
13523074	Ahsan Malik Al Farisi	CCManager, LogHandler & UnitTest
13523026	Bertha Soliany Frandi	ValidationBased Algorithm, Transaction & Action
13523056	Salman Hanif	LockBased Algorithm
13622076	Ziyan Agil Nur Ramadhan	MVCC Algorithm

4. Storage Manager

NIM	Nama	Workload
13523040	Kenneth Poenadi	Implementasi read block, membantu integrasi b+ tree index, hash index
13523072	Sabilul Huda	Implementasi delete blocks dan b+tree index
13523098	Muhammad Adha Ridwan	Implementasi Hash Index

13523116	Fityatul Haq Rosyidi	Implementasi Write block dan bonus drop table

5. Failure Recovery Manager

NIM	Nama	Workload
13523032	Nathan Jovial Hartono	Inisialisasi struktur Class dan implementasi fungsi write_log
13523044	Muhammad Luqman Hakim	Inisialisasi struktur dan desain Class
13523034	Rafizan Muhammad Syawalazmi	Implementasi transaction recovery dan system crash recovery
13523012	Felix Chandra	Implementasi Unit Testing, debugging dan perbaikan kode modul Failure Recovery Manager

Lampiran

Release:

https://github.com/pixelatedbus/database-management-system-development-IF3140/releases/tag/v4.1