Code Generation Recovery Log Replay for In-Memory Database Management Systems

Tianlei Pan

June

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Andy Pavlo, Chair
Wenting Ye

Submitted in partial fulfillment of the requirements for the Fifth Year Master's Program.

Keywords: Code Generation, Log Replay, Recovery, query compilation

Abstract

Code generation is an optimization technique for improving query execution throughput by compiling query plans into native code. This technique, however, leads to design challenges for the recovery system of a database management system. The log replay process will be disjoint from the built-in execution engine that has adapted to operate efficiently on compiled code. This leads to the implementation of a separate execution engine to deal with the execution of log records, which can be a huge waste of engineering efforts. To resolve this design conflict between code generation and database recovery, we present a unified approach to support both query execution and log replay in a code-generation-based DBMS. We ask the recovery system to convert log records into compiled code that will be easily accepted by the execution engine. Our results show that while our approach incurs a higher performance overhead compared to using a separate execution engine, it requires much less engineering effort and is superior in index updates.

Acknowledgments

yes

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management system (DBMS) is a category of software that are responsible for storing data, analyzing data, and interacting with applications. In 1970, E.F. Codd proposed the relational model of data,

Depending on the use case, a DBMS can focus more on either capturing data or analyzing data. On-line Analytical Processing (OLAP) DBMSs focus on reading, analyzing and aggregating data that is less likely to be modified. On the other hand, On-line Transaction Processing (OLTP) DBMSs support write-heavy transactions that modify the database frequently [6].

Regardless of the type of a DBMS, both OLTP and OLAP DBMSs are susceptible to failures. The DBMS may shutdown unexpectedly, fail to execute a query, or cease to function due to corrupted data. These failures threaten the integrity of the database and make the DBMS unreliable. Therefore, it is important for a DBMS to distinguish between different kinds of failures in order to develop mechanisms to maintain integrity.

A DBMS can encounter three types of failures: Transaction Failures, System Failures, and Media Failures[17]. Transaction Failures are the most common type of DBMS failures. Transaction Failures occur when a transaction fails to commit, either at its own request (e.g., logical errors) or on behalf of the DBMS (e.g., resource unavailability). When a transaction fails, the system needs to react within its lifetime to prevent data inconsistency [24]. The DBMS needs to decide whether and how it should undo the changes of the failing transaction. Since a DBMS is running transactions rapidly while its active, Transaction Failures can occur as much as 100 times per minute[17]. System Failures occur due to hardware failures (e.g., power outage), operating system faults (e.g., insufficient memory) or DBMS exceptions. Each of these trigger events can cause the DBMS to shutdown unexpectedly and uncontrollably. System Failures occur less frequently compared to Transaction Failures, but take a much longer time for the DBMS to recover. The database needs to pinpoint time of failure and what changes the DBMS failed to record. Media Failures [28] happen when the underlying storage device of a DBMS fails (e.g., disk head crash, bad sectors). This type of failure causes permanent data loss to the DBMS. The only way for a DBMS to recover from Media Failures is to restore from a secondary backup storage.

To handle different types of failures, a DBMS needs to develop a recovery system that will restart the database correctly in case of failure. A widely adopted recovery system is to combine write-ahead logging during transaction execution with crash recovery that uses redo (install changes to the database) and undo (remove changes from the database) processes, represented by ARIES[22]. The ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) [24] protocol is a recovery method developed by IBM in the 1990s. It guarantees database integrity in the fact of Transaction, System and Media Failures [24]. In ARIES, the DBMS records transaction modifications in durable log records before the DBMS propagates changes to a database page to disk. During crash recovery, the DBMS applies changes from the log records to the database with redo and undo processes (i.e., log replay) [12].

However, the original ARIES paper focused on disk-based DBMSs. Recently, we have seen a rapidly increasing number of in-memory DBMSs due to technological advancement in semiconductor memory [14]. Many in-memory DBMSs avoid strictly adhering to the ARIES protocol for recovery. This is due to the fact that a lot of the concepts in ARIES no longer apply to in-memory DBMSs. For instance, ARIES uses undo records to revert changes that have been applied to the database. However, undo records are excessive for many in-memory DBMS recovery systems [5, 8, 15, 22, 29]. Performance concerns is another reason why in-memory DBMSs stray from the ARIES protocol. Logging requires the DBMS to interact with the disk. Therefore, logging I/O is a major bottleneck for an in-memory DBMS. This prompts in-memory DBMSs to minimize logging traffic [11]. An example is the recovery of database indexes [11]. Disk-based DBMSs log updates to index structures (e.g., B+ Tree) that allow faster recovery during log replay [24]. In comparison, in-memory DBMSs do not log index updates [5, 8, 22, 29] and choose to reconstruct indexes from scratch during log replay. Despite the differences, many in-memory DBMSs still use some form of logging [5, 8, 22, 29] in their recovery systems.

In addition to changes in recovery systems, in-memory DBMSs adopt various optimization algorithms for their execution engines[18]. These optimizations aim to increase execution throughput by reducing either the number of instructions that a DBMS executes to run a query, or the clock cycles per instruction (CPI) [4]. Query compilation/code generation is an important optimization technique used by DBMSs to greatly reduce the number of instructions that the CPU needs to execute[18]. During execution, the DBMS breaks down a query into various tasks [5, 25]. To speed up the execution of those tasks, the DBMS can compile them into native code (e.g., C/C++) with an off-shelf compiler[4]. Query compilation leads to faster query execution because it specializes both the data structures (e.g., hash table) and access methods of a DBMS towards execution efficiency [25]. Moreover, the compiled code can be optimized around locality that increases the chance of a data tuple being propagated between operators in CPU registers[23].

While query compilation succeeds in increasing the execution throughput of a DBMS, it comes with several drawbacks. Implementing a query compilation system requires additional knowledge of compiler systems (e.g., LLVM) and huge amounts of engineering work to translate different execution tasks[18]. The DBMS also generates low-level machine code that is hard to understand and debug [18].

Moreover, the DBMS recovery system is isolated from the query compilation execution engine (i.e., a component of a DBMS that is responsible for execution). Most of the query compilation DBMSs we have surveyed [5, 13, 25, 26] have no conduits between their recovery systems and execution pipelines. The recovery system implements its own functionalities to update the database, but some of these functionalities already exist in the execution engine. For instance, we mentioned that in-memory DBMSs rebuild the indexes during log replay. When a redo process install changes on a table, it also retrieves the indexes on that table and update them (index update). However, the execution engine has already implemented this index update functionality. As a result, the DBMS contains two implementations that provide identical functionalities. The number of those functionalities in the recovery system will eventually amount to the scale of a separate execution engine that is built specifically for log replay.

Therefore, if we can find a way to unify a DBMS's recovery system and execution engine, we can drastically reduce engineering overhead. Furthermore, we believe this unification also increases replay efficiency. Unless the rewritten functionalities in the recovery manager are executed as native machine code, they will be less efficient compared to those in the execution engine that uses query compilation. Therefore, if the DBMS allows the recovery system to invoke functionalities within the execution engine, the log replay process should be more efficient.

1.1 Contribution

We present a system design that will extend the execution engine to create custom programs for replaying physical log records. Our design converts log records into a format that the DBMS can compile into native code to be accepted by the execution engine. We will show that our approach removes the need for introducing extra implementations into the recovery system for functionalities that already exist in the execution engine. We will also verify our assumption that by utilizing the code generation execution engine, the DBMS will have increased log replay efficiency.

We implement our approach in NoisePage[5], a self-driving in-memory DBMS developed at Carnegie Mellon University. NoisePage is built in C++ and uses the PostgreSQL wire protocol for user communication. The DBMS also depends on query compilation for its execution engine. Our experimental results show that while the code generation recovery approach falls short of recovery throughput compared to baseline recovery, it is more efficient in certain areas (e.g., index maintenance, table constraints) and has higher scalabiltiy for a table with a larger number of columns.

We structure the remainder of the thesis as follows. In Chapter 2, we provide more background information on database recovery, code generation, and motivations behind our approach. In Chapter 3, we present the recovery and code generation architecture in NoisePage and how we implemented our approach in this DBMS. In Chapter 4, we evaluate our implementation by comparing it against NoisePage's builtin recovery system. We discuss related works in Chapter 5 and conclude our thesis in Chapter 6.

Chapter 2

Background

In this chapter, we discuss more background information on database recovery and code generation. We then explain how the two components interact within a DBMS and what changes need to be made for them to cooperate.

2.1 Database Recovery

A DBMS ensures database integrity by satisfying the following conditions:

- **Durability of Updates** [1]: All the changes made by committed transactions are durable.
- **Failure Atomicity** [12]: None of the changes made by aborted or failed transactions are persisted visible after recovery.

Many DBMSs [2, 5, 8, 9, 10, 22, 29] combine logging and recovery protocols to preserve these two conditions in case of failure. Logging is the action of storing information about committed transactions on disk, while recovery is the action of restoring the database system into a consistent state.

The DBMS stores logging information in a special data structure called a log record. A log record contains physical changes (e.g., changes made to a specific physical address) performed on the DBMS, or higher-level information (e.g., user-input query). In write-ahead logging [24], the DBMS records physical database changes in a log file before the DBMS persists the changes on disk. A transaction is not allowed to commit until the DBMS persists its corresponding log record.

For disk-based DBMSs, a log record in the log file can be either a redo record or an undo record [24]. During recovery, a redo record installs the effects of committed transactions, while an undo record removes the effects on incomplete or aborted transactions. In-memory DBMSs, on the other hand, do not persist uncommitted changes on disk [5, 8, 15, 22, 29]. This is because disk-based DBMSs generate dirty pages. These are pieces of modified entries in the database that reside in the memory, but have not yet been persisted to disk by the DBMS. However, in-memory DBMSs do not write dirty data to persistent storage [5, 8, 15, 22, 29]. Therefore, in-memory DBMSs do not generate dirty pages, nor do they need to keep undo records. As a result, undo records are no longer required for their recovery systems. Moreover, in-memory DBMSs do not need to generate log records for indexes [11]. During recovery, the DBMS reconstructs

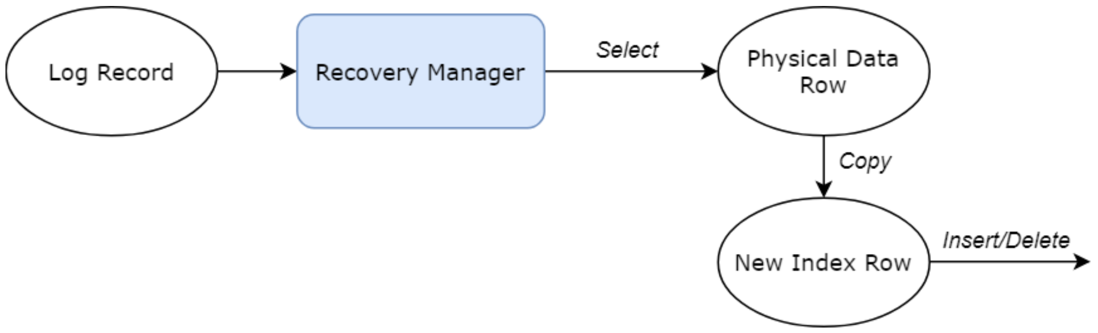


Figure 2.1: Index Update During Recovery.

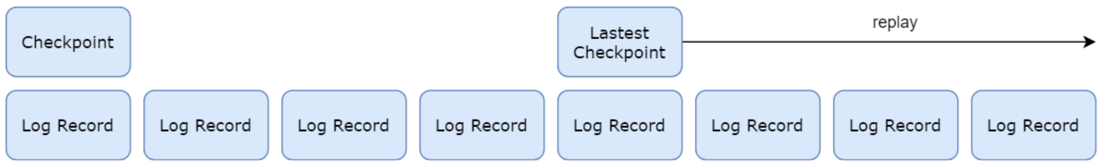


Figure 2.2: Recovery with Log Records and Checkpoints.

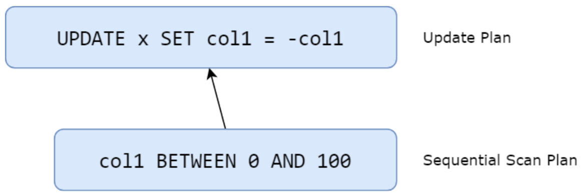


Figure 2.3: Physical Plan that corresponds to **`UPDATE x SET col1 = -col1 WHERE col1 BETWEEN 0 AND 100`**.

the database with redo records reconstructs and create the indexes simultaneously [8, 22]. For each redo record, the recovery system finds the table that it modified, then retrieves all of its indexes from the database catalog ((Figure 2.1). Finally, for each index, the recovery system either updates it with new values, or remove entries from the index.

Aside from logs, recovery systems can store complete snapshots (i.e., checkpoints) of the database. The DBMS can decide to take checkpoints based on a certain rule (e.g., timed-interval, number of transactions). Checkpoints reduce the time required for the database to recover. The DBMS can directly revert to a previous snapshot without replaying any log records before that snapshot. DBMSs often store both log records and checkpoints. During recovery, the DBMS finds its most recent snapshot, then replays any log records that are stored after that snapshot (Figure 2.2).

2.2 Query Compilation / Code Generation

When a query arrives, the DBMS parses the query into an abstract syntax tree (AST). The optimizer of the DBMS then converts the AST into a physical plan tree. The physical plan represents how the DBMS will execute the query. It consists of operators that specify physical operations

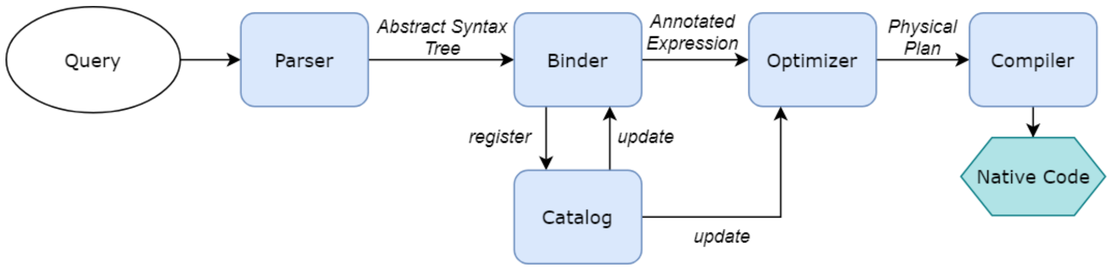


Figure 2.4: Query Compilation Architecture.

on the DBMS (e.g., inserting values into a certain memory location). For instance, **`UPDATE x SET col1 = -col1 WHERE col1 BETWEEN 0 AND 100`** can be represented by an update plan tree with a sequential plan as a child (Figure 2.3). The execution engine then uses the plan tree for execution.

In a DBMS that uses query interpretation, its execution engine (i.e., a component of a DBMS that is responsible for execution) processes the physical plan by traversing the plan tree. Therefore, for every query, DBMS needs to follow pointers and resolve branching conditions (e.g., if, switch statements). This incurs overhead caused from virtual function calls, branch mispredictions and instruction cache misses.

Query compilation eliminates the cost of query interpretation by compiling the physical plan to machine code that is specific for that query. The machine code of the corresponding physical plan can then be executed repeatedly by the execution engine, removing any need for virtual function calls and branching resolutions. Generally, there are two ways of compiling queries. In the first approach, the DBMS creates source code that is compiled into native code with an external compiler (e.g., gcc) [19, 20]. This approach is used in Amazon Redshift and pre-2016 MemSQL [26]. In the second approach, the DBMS first converts physical plan into an intermediate representation (e.g., LLVM) that follows the grammar of an imperative language. This approach simplifies the process of converting the physical plan into machine code, as the intermediate representation form is designed to resemble SQL statements. Moreover, it removes the need for the DBMS to use the compiler as an external process. This approach is adopted by NoisePage [23], Hyper [25], Hekaton [13], and SingleStore [26].

2.3 Motivation

While code generation is a powerful optimization technique, it poses design challenges for the recovery system. Both the execution engine and physical log records perform identical physical operations on a database table. The native code generated through query compilation operates directly on physical tables. For physical log records, they are already designed as a data structure that represents physical changes made to the database. During recovery, the DBMS simply replays those changes on the table.

However, the recovery system has no means of reusing functions within the execution engine to install physical changes from log records. The execution engine does not accept physical log records because it follows along the query compilation pipeline. This pipeline transforms

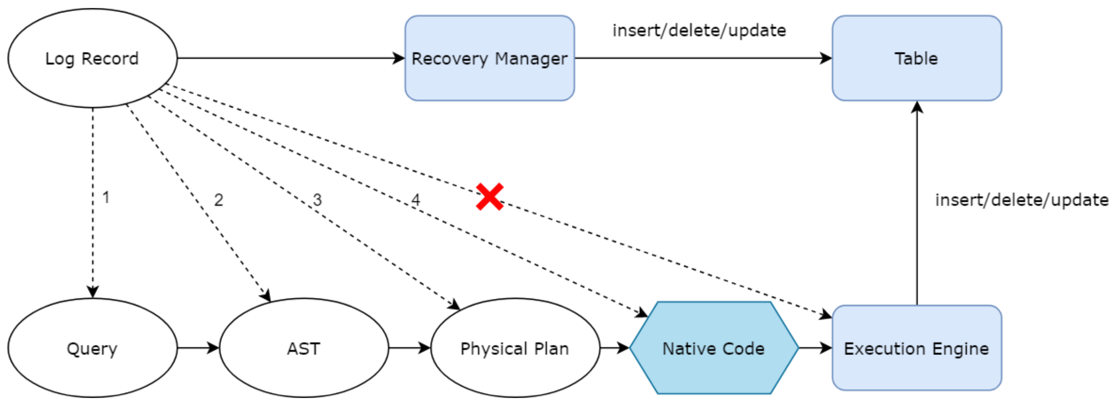


Figure 2.5: Possible ways to integrate a physical logging system with code generation.

one logical representation of a query to another, until the query becomes native code. There is no place for physical log records to exist within this transformation pipeline.

There are two solutions to this problem. One solution is to re-implement functions to install physical changes on data tables within the recovery system. However, this approach completely isolates the recovery system from the execution engine. This means that if the recovery system wishes to use a functionality within the execution engine, it has to rewrite that same functionality within the recovery system (**??**).

The other solution is to integrate physical logging with the execution engine. Figure 2.5 shows four possible paths that allow the execution engine to accept physical log records. Converting the log record directly into native code requires the implementation of a new compiling framework, which makes this option infeasible. Therefore, the closest point where the log records can reach before entering the execution engine is physical plans. We then reach the conclusion to convert log records into physical plans. Firstly, compared to raw queries or abstract syntax trees, physical plans are closer to the end of the query compilation pipeline. If we convert a log record into an abstract syntax tree (AST), the DBMS still needs to do additional work to convert the AST into a physical plan. Secondly, we will show in later sections that it is enough for physical plans to represent contents of a log record.

If the DBMS can convert log records into physical plans, the recovery system can rely on code generation for recovery operation. This code generation recovery approach brings around several benefits. Firstly, it significantly reduces engineering overhead. Compiling contents of the log records into machine code allows the recovery system to seamlessly integrate with the built-in execution engine. We have explained in Section 2.1 that the recovery system needs to perform extra operations (e.g. index updates in **??**) besides updates to the data tables. These functionalities exist within the execution engine. If we allow the recovery system to reuse existing functionalities by communicating with the execution engine through native code, then the recovery system no longer needs to implement its own functionalities. Secondly, utilizing the code generation execution engine provides more efficient functionalities. A function implemented by the recovery system will not be as efficient as one that exists in the execution engine.

These potential benefits motivate us to design a code generation recovery system based on conversion from log records to physical plans. In the next chapter, we will present the implementation of our approach .

Chapter 3

Method

We now describe how to extend the recovery system and query compilation pipeline to support log replay with code generation. Our implementation follows along the third option shown in Figure 2.5. We first enable the recovery system to convert log records into compiled native code. To achieve this, the DBMS needs to interpret the log records into physical plans and prepare them for compilation. We then need to make minor modifications to the execution engine, so it will be able to correctly execute the native code generated by the recovery system. We further summarize our approach (Figure 3.1) in two steps:

1. Convert the log records into physical plans that can be compiled into native code (Recovery Code Generation).
2. Run the compiled machine code using the execution engine (Recovery Execution).

We implement our approach in NoisePage [5]. Noisepage depends on a recovery system that uses physical logging schemes. It also adopts query compilation optimization for its execution engine.

3.1 Recovery System Architecture

To show how we will extend the recovery system, we first present some background information on the recovery system architecture of NoisePage. NoisePage's recovery system consists of three major components: Transaction Manager, Logging Manager, and Recovery Manager. The Logging Manager interacts with the Transaction Manager to achieve physical logging, while the Recovery Manager interacts with the Transaction Manager to provide database recovery.

3.1.1 Transaction Manager

The Transaction Manager is responsible for creating and maintaining database transactions. NoisePage uses a transaction to perform physical updates on the database and physical logging to disk. NoisePage uses a multi-versioned delta store [27] Transaction Manager that ensures `SNAPSHOT-ISOLATION` [3]. The Transaction Manager allows non-blocking reads over writes and vice versa, but does not allow write-write conflicts on a per tuple basis. A transaction's lifecycle

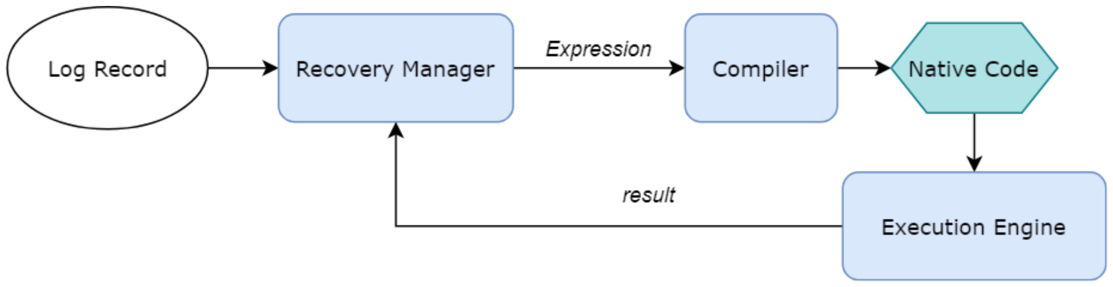


Figure 3.1: Code Generation Recovery Mechanism.

starts when the transaction begins, and ends after the log manager has serialized its changes to disk.

Each data tuple in NoisePage is uniquely identified through a tuple slot that stores the offset of a tuple in a data block. When the DBMS inserts a tuple into a table, it creates a new tuple slot that points to a memory location within the table's data block. Subsequent updates to the tuple does not create direct copies of the tuple, but stores delta information (redo records) about the updates instead. These redo records are essential for executing log replay correctly. The structure of a redo record is shown later in Figure 3.4.

A transaction uses a buffer (redo buffer) that stores all the delta records that will be generated in its lifetime. Each redo buffer has a fixed size and is allocated from a centralized memory pool. Each time a transaction needs to change the contents of the DBMS, it will attempt to request space from the redo buffer to store a corresponding delta record of the change. If the redo buffer runs out of space, the transaction will replace the current buffer with a new buffer from the memory pool. The redo buffer allows the logging components to process the changes before the transaction ends.

When a transaction commits, the Transaction Manager creates a commit record that contains a timestamp that refers to the oldest active transaction from the timestamp manager. The commit record is appended to the redo buffer. The redo buffer is then carried over to the log manager, where the changes will be serialized to disk. When a transaction aborts, the transaction creates an abort record that prevents corresponding records to be replayed during recovery. Abort records are essential to recovery; since the redo buffer is persisted to disk by the log manager once it is full, it is possible for aborted transactions to persist on disk. During recovery, the DBMS needs to remove aborted transactions. Aborted transactions that are persisted on disk are rare [7], however, since it would be difficult for an OLTP transaction to fill up the entire redo buffer.

3.1.2 Log Manager

The Log Manager (Figure 3.2) is responsible for performing write-ahead logging [24] in NoisePage. Each log record is self-contained and does not require any additional metadata to be replayed. The Log Manager consists of two separate tasks that live in different threads: the Log Serializer tasks and the Log Consumer tasks.

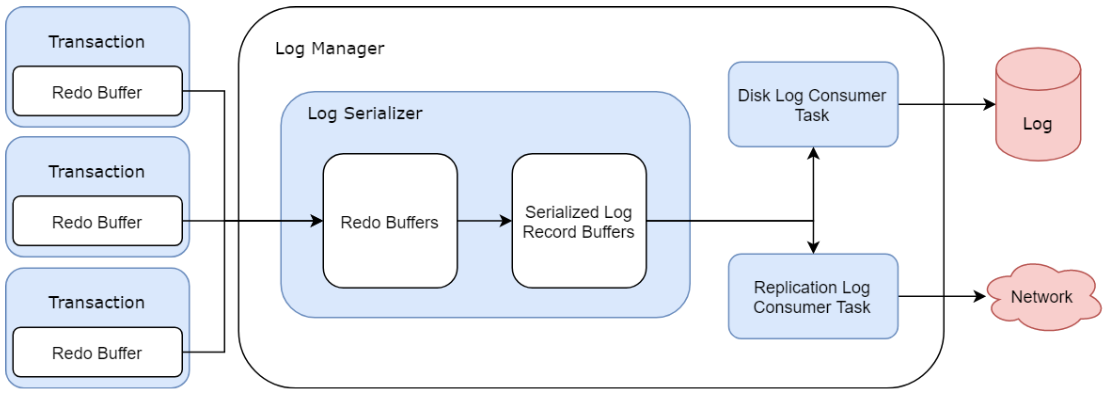


Figure 3.2: Log Manager.

Log Serializer

The Log Serializer task receives redo buffers from a transaction once it is full, or when the transaction decides to commit. The Log Serializer breaks down the redo buffers into raw memory segmented into fixed-size buffers. The Log Consumer consumes these buffers for persistence.

The DBMS must serialize log records in the correct order for recovery to perform correctly. Different transactions can share a same redo buffer that is filled with each of their own log records. Moreover, it is possible for transactions to appear in a non-serial order relative to their begin timestamp. However, the log records performed by an individual transaction is guaranteed to appear in the same order as they were created. This property helps ensure `SNAPSHOT-ISOLATION` property of NoisePage during log replay.

Log Consumer

Different Log Consumers are responsible for consuming buffers supplied by the Log Serializer. The Log Serializer provides each Log Consumer a copy of the original buffer to achieve parallelism. NoisePage uses two type of log consumers: Disk Log Consumer and Replication Log Consumer.

The Disk Log Consumer repeatedly polls for new buffers from the log serializer and writes the changes to disk. To improve performance, disk log consumer uses group commit that allows a batch of changes to be committed over a time period specified by the user. Under default settings, the disk log consumer writes down changes every 10 microseconds of if more than 1 MB of data has been written since the last write.

The Replication Log Consumer also polls for new buffers. Instead of persisting them to disk, it sends the new buffers over across the network to database replicas.

3.1.3 Recovery Manager

Figure 3.3 shows the architecture of the Recovery Manager that is responsible for log replay. NoisePage uses four different record types: redo, delete, commit, abort. The Recovery Manager do not process redo and delete records are immediately. Instead, the Recovery Manager place

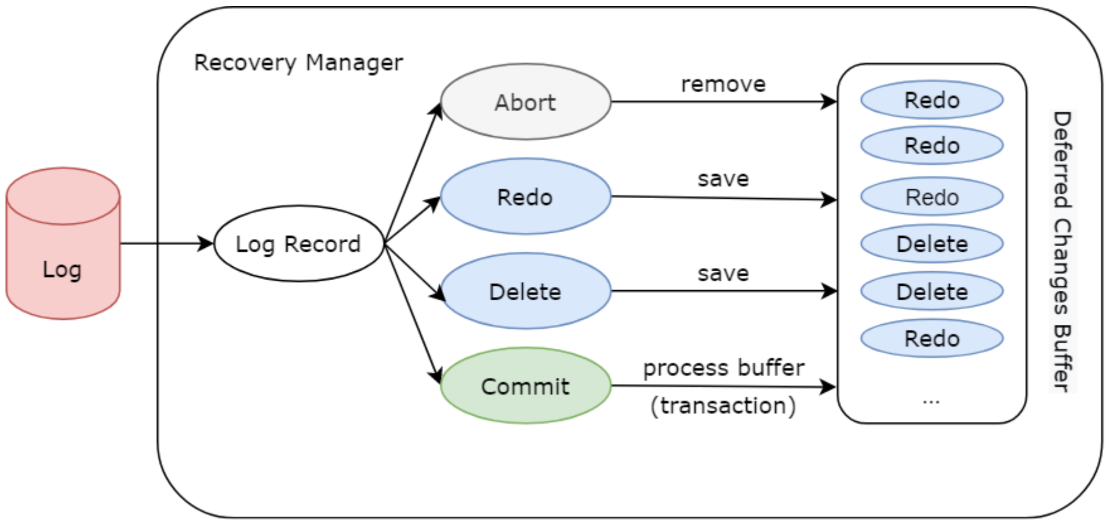
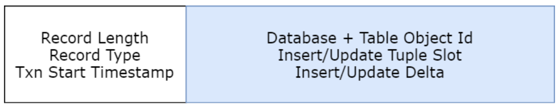
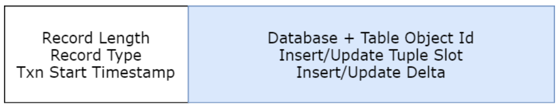


Figure 3.3: Recovery Manager.



(a) Redo Record Structure.



(b) Delete Record Structure.

Figure 3.4: Log Record.

redo and delete records inside a deferred changes buffer, ordered by their transaction timestamp. An abort record removes a corresponding redo/delete record from the buffered. A commit record initiates a transaction to iterate over the deferred changes buffer. Once all the Recovery Manager processed all the deferred changes, it clears the buffer and commits the transaction. The transaction aborts if any log record is malformed or any replay fails.

Log Record

A physical log record consists of a header, followed by a record body (Figure 3.4). The header stores metadata about the log record itself. This includes a log record's record type, size, and begin timestamp of the transaction (See Section 3.1.1). The record body contains physical information that is necessary for recovery and varies depending on the record type. Depending on the value type, the values may bee stored in-line or not in-inline. Simple types such as `INTEGER`, `FLOAT` can be stored in-line, while `VARLEN` entry may store a pointer in the values column that points to another physical location in the record.

Tuple slots in the log records will no longer be valid memory locations during recovery. Instead, the Recovery Manager creates an internal mapping from original tuple slots to new tuple slots that will be retrieved after new values are inserted. The tuple slot mapping allows the Recovery Manager to apply changes to the correct memory locations in the new memory environment.

While storing tuple slots is sufficient for delete records, redo records require additional data

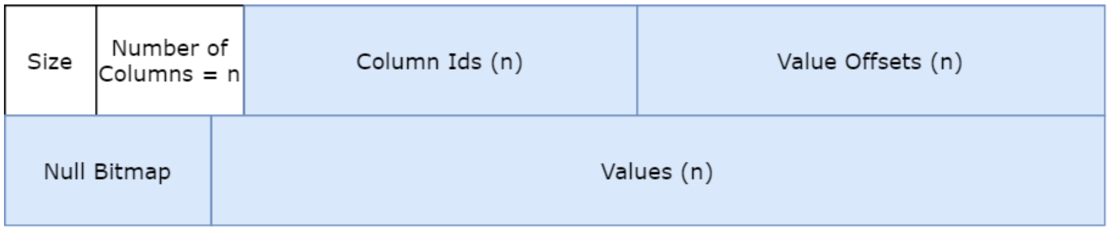


Figure 3.5: Redo Record Projected Row.

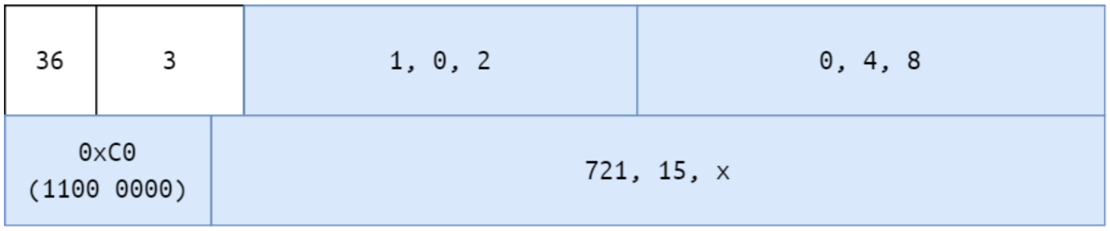


Figure 3.6: Projected Row that represents [15, 721, `NULL`].

to perform log replay. For instance, to perform an insert, the redo record must contain the values to insert. The Log Manager stores these data in a log record's projected row (Figure 3.5). A projected row represents a database row. The DBMS can access an individual value within the projected row by identifying a column id and reading its value using that column's value offset Figure 3.6.

Replay Step

A replay step refers to the recovery manager replaying a single redo record or delete record. A NoisePage replay step has three different scenarios:

- **Insert Replay**: The Recovery Manager inserts the values stored within a redo record into the given tuple slot and updates the indexes.
- **Delete Replay**: The Recovery Manager deletes the tuple slot specified by a delete record and updates the indexes.
- **Update Replay**: The Recovery Manager updates the tuple slot using the values within a redo record and updates the indexes.

While both insert and update redo records follow the same redo record structure, the Recovery Manager differentiates between them by keeping track of all the tuple slots found in log records. If the Recovery Manager has never seen a tuple slot inside a redo record before, then the redo record represents an Insert operation. Otherwise, the Redo Record represents an Update operation.

3.2 Recovery Code Generation

In this section, we introduce how we integrate the NoisePage's recovery system with its code generation pipeline. We first provide an overview over NoisePage's code generation pipeline.

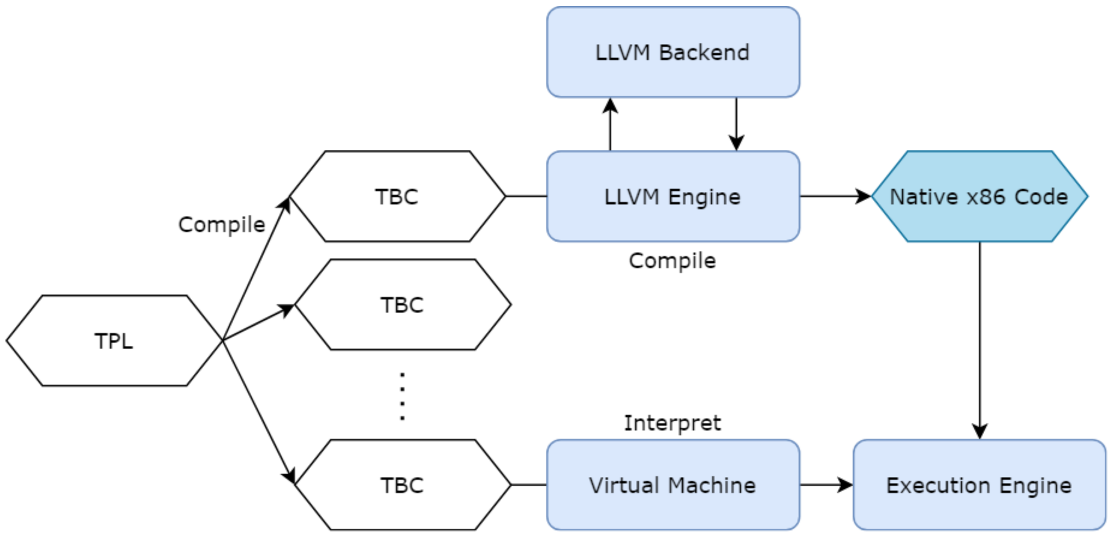


Figure 3.7: Lifecycle of a TPL code fragment. Execution can run in either interpret or compile mode.

We then show how we convert redo and delete records into physical plans that can be accepted by the code generation pipeline for compilation.

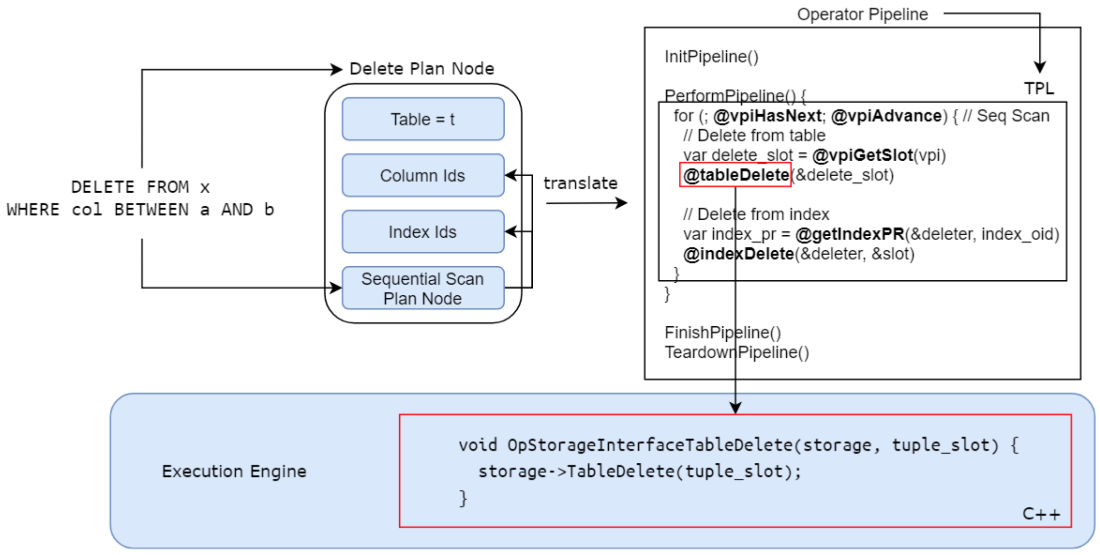
3.2.1 Code Generation Pipeline

The code generation pipeline in NoisePage follows from the data-centric compilation approach in [25], as seen in Figure 2.4. The DBMS interprets a query, parses into an AST and converts it into a physical plan tree. A plan tree consists of plan nodes and specifies how the query should be executed on the physical level of the database.

A plan translator then converts a physical plan tree/node into an operator pipeline. Each plan translator correspond to a plan node in the original plan tree. A pipeline contains an ordered collection of relational operators that are not yet materialized, ended with a pipeline breaker (e.g., sort tuples, creating a hash table). The pipeline is only materialized whenever a pipeline breaker has been encountered.

NoisePage then compiles each operator pipeline into a customized imperative language (TPL). The DBMS then compiles each function of a query inside TPL into some bytecode (TBC). The bytecode can be interpreted by a built-in virtual machine, or further compiled into an LLVM module. The execution engine can either interpret TBC, or run the compiled LLVM module (Figure 3.7).

We further demonstrate the process of query compilation through the example in Figure 3.8. To run the query `DELETE FROM x WHERE col BETWEEN a AND b`, the DBMS needs to first do a sequential scan to find out the columns, then do a delete operation on the found columns. The DBMS represents these two operations in a delete plan node that includes a sequential scan plan node as a child. The DBMS then translates the plan node into an operator pipeline that specifies how each operation will be exactly executed in the DBMS. After this, the DBMS compiles the operator pipeline into TPL code. Each function inside the TPL code



```
DELETE FROM x
WHERE col BETWEEN a AND b
```

```
InitPipeline()

PerformPipeline() {
  for (; @vpiHasNext; @vpiAdvance) { // Seq Scan
    // Delete from table
    var delete_slot = @vpiGetSlot(vpi)
    @tableDelete(&delete_slot)

    // Delete from index
    var index_pr = @getIndexPR(&deleter, index_oid)
    @indexDelete(&deleter, &slot)
  }
}

FinishPipeline()
TeardownPipeline()
```

```
void OpStorageInterfaceTableDelete(storage, tuple_slot) {
  storage->TableDelete(tuple_slot);
}
```

Figure 3.8: Code Generation of `DELETE FROM x WHERE col BETWEEN a AND b`

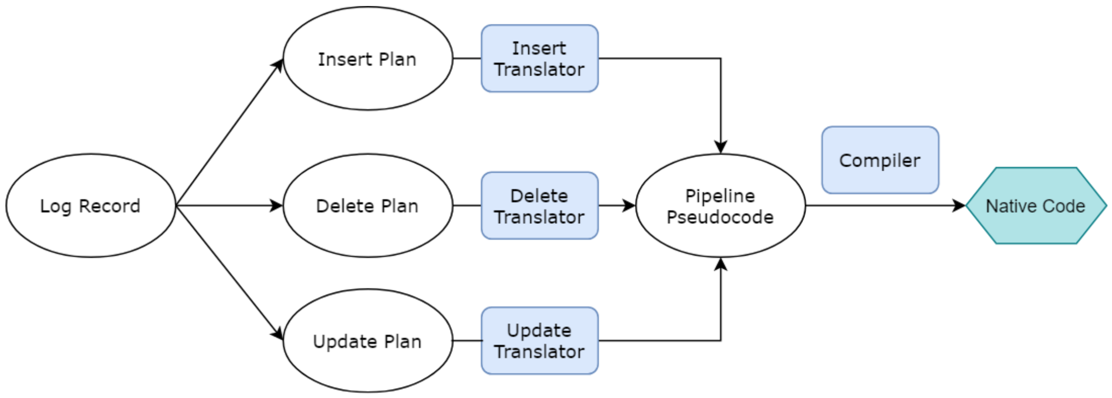


Figure 3.9: Process for converting log records into native code.

correspond to a piece of function inside the execution engine that will be evetually invoked once the compiled native code is run. For instance, `@tableDelete` corresponds to a C++ function `OpStorageInterfaceTableDelete` (a delete operation on a tuple slot of a table) in the execution engine.

3.2.2 Recovery Integration

We can consider each type of replay as a form of SQL statement that can be converted into native code through the code generation process discussed above (Figure 3.9). We further divide the code generation process for a log record into three steps: plan generation (creating the physical plan node), plan translation (translating the plan into operator pipelines) and compilation. In plan generation, the Recovery Manager converts each replay process (Insert Replay, Update Replay, Delete Replay) into a corresponding plan node. In plan translation, the Recovery Manager uses matching plan translators to convert physical plans into operator pipelines.

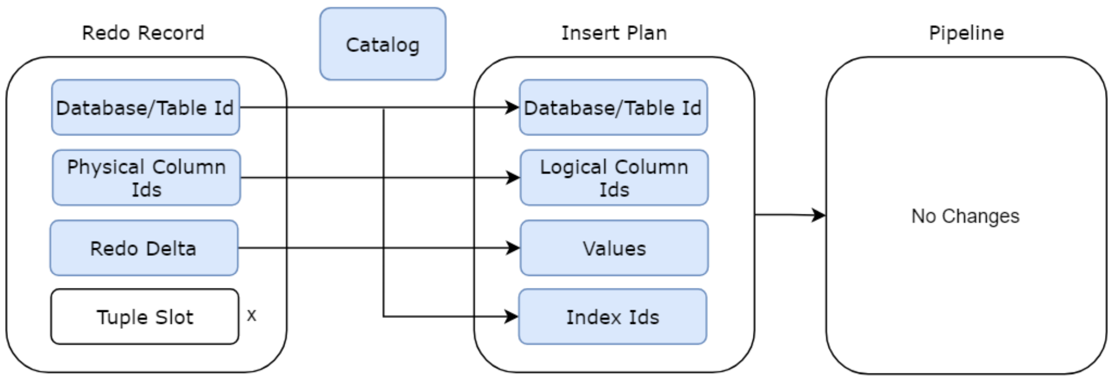


Figure 3.10: Conversion from an Insert Redo Record into an Insert Plan Node. The tuple slot provided by the redo record is discarded.

Once the Recovery Manager has the operator pipeline, it can then use the compiler to generate machine code for execution.

Insert Replay Conversion

Insert plan nodes correspond to **`INSERT`** statements. We can represent an insert redo record as the SQL statement **`INSERT INTO x VALUES y`**.

(Figure 3.10) shows the conversion from an insert redo record into an insert plan node. The Recovery Manager can retrieve metadata (e.g. table id, column ids) from the redo record and the database catalog. The redo record's projected row contains values for insert (Figure 3.5). Those values are in primitive forms (e.g., integer, float, double) as constants. The Recovery Manager can then feed those constants into the insert plan node. This conversion requires no changes to the plan nodes, nor to the operator pipelines.

Delete Replay Conversion

Delete plan nodes correspond to **`DELETE`** statements. **`DELETE`** statements require specifications on where the DBMS should perform delete operations (e.g., **`WHERE`** clause in Figure 3.8). Therefore, a delete plan node requires a child plan node to function. It uses its child plan node to figure out which tuple slot to schedule for deletion.

However, a delete record's tuple slot, combined with the tuple mapping from Recovery Manager, already points to the tuple slot to delete. In this case, there will be no need for the plan node to go through the child node and find out the delete tuple slot. Moreover, delete records do not contain value information that is required for index update (Figure 3.4). Therefore, we need to provide the operator pipeline with a projected row that contains values correspond to the row of the tuple slot from the Recovery Manager.

We change the behavior of the operator pipeline as follows: (1) remove the step to look for a new delete tuple slot, (2) retrieve the delete slot supplied by the Recovery Manager during execution (Figure 3.11), and (3) use the projected row provided by the Recovery Manager for index updates.

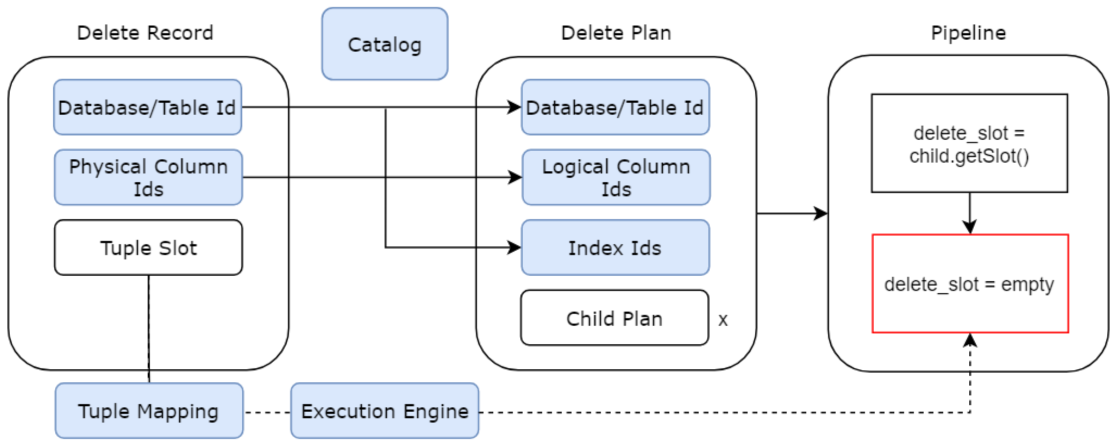


Figure 3.11: Conversion from a Delete Redo Record into a Delete Plan Node. The operator pipeline uses the tuple slot from the delete record.

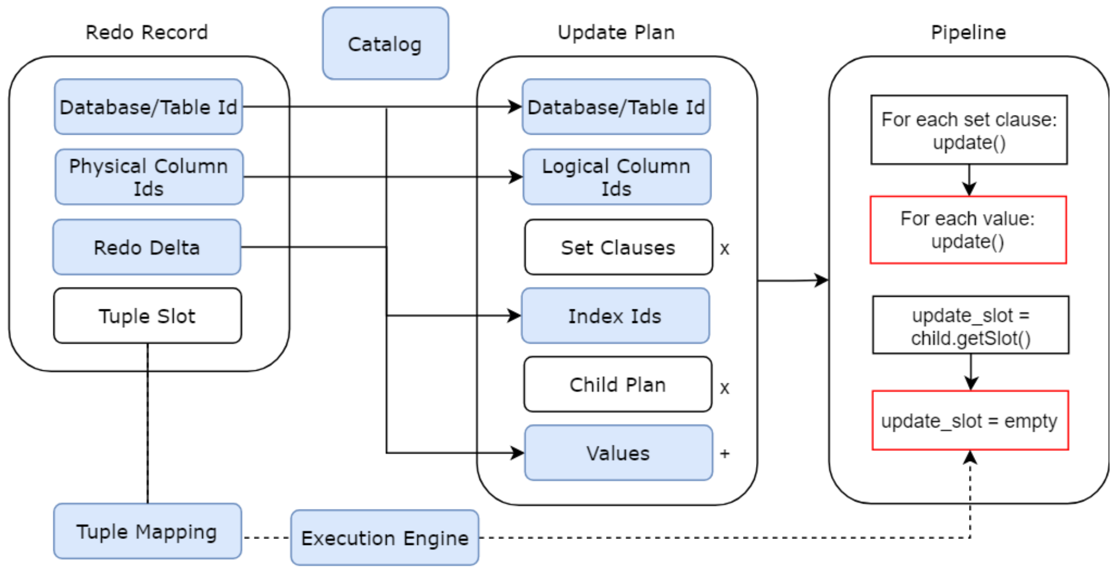
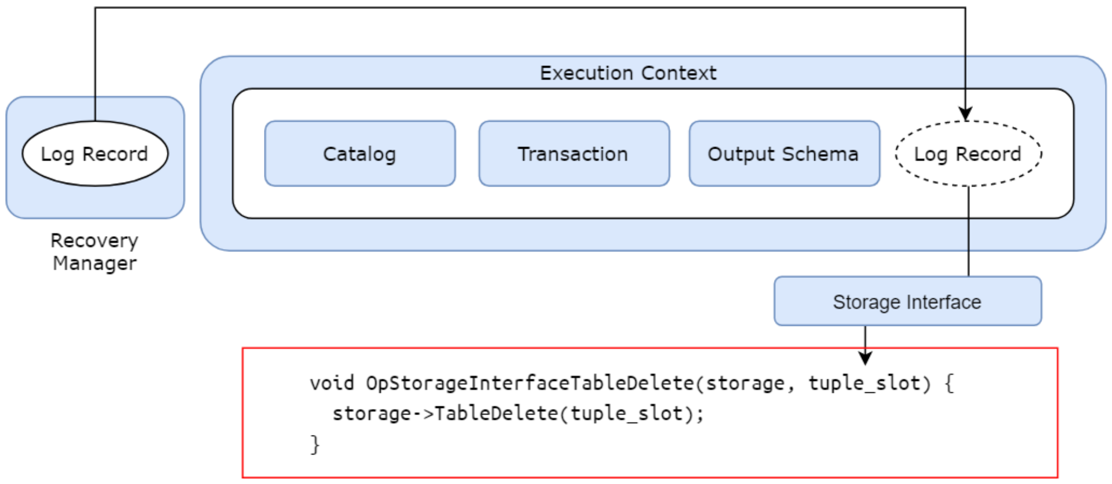


Figure 3.12: Conversion from an Update Redo Record into an Update Plan Node. **SET** clauses and child plan nodes inside the update plan node are discarded.

Update Replay Conversion

Update plan nodes correspond to **UPDATE** statements. Similar to **DELETE** statements, update plan nodes require child plan nodes. An update operator pipeline expects **SET** clauses returned from the child node (e.g., sequential scan). These **SET** clauses, however, describe higher-level logical operations (i.e., update a column with a certain value). On the other hand, redo records contain low-level physical information (i.e., update a tuple slot with a certain value). Therefore, the Recovery Manager cannot create **SET** clauses efficiently from redo records.

We therefore perform the following changes: (1) extend the update plan node to accept primitive values like an insert plan node does, (2) modify the operator pipeline to process the



```
void OpStorageInterfaceTableDelete(storage, tuple_slot) {
  storage->TableDelete(tuple_slot);
}
```

Figure 3.13: Operation code functions can access log records through an Execution Context.

primitive values instead of **SET** clauses during a replay step (Figure 3.12), (3) remove the step for update operator pipeline to find out the update tuple slot, and (4) retrieve the update tuple slot from the recovery manager.

3.3 Recovery Execution

We explained in Section 3.2.2 that we modified deletes/update operator pipelines to retrieve certain objects (e.g., tuple slots, projected row) from the Recovery Manager. To achieve this, we need to modify the execution engine. The execution engine is responsible for running the compiled query. Inside the execution engine, the compiled query invokes different operation codes (e.g., `OpStorageInterfaceTableDelete` in Figure 3.8) that correspond to operators in the operator pipeline. Since we modified the operator pipelines earlier, we now need to change operation codes to read in objects from the Recovery Manager.

3.3.1 Execution Changes

We first use the Recovery Manager to set the required tuple slots and projected row into a log record. Before the execution engine runs the query, the Recovery Manager needs to setup a context for execution. In NoisePage, an execution context encapsulates information that is supplied by upper layers of the DBMS (e.g. access to the database catalog, the transaction running the query, memory pool). We pass the log record from the Recovery Manager into the execution context.

During recovery, for each operation code function that needs to use predetermined tuple slots/projected row for delete/update due to our changes in the operator pipelines, we enforce it to access those values from the execution context (Figure 3.13). During the execution of operation codes, NoisePage uses a storage interface object to manipulate over a certain DBMS table. The execution engine initalizes the storage interface object with the execution context.

Therefore, the storage interface provides access for the operation codes to access information in the log record from the Recovery Manager.

3.4 Caching Optimization

The naive implementation for code generation searches for table metadata and re-compiles the native code every time the Recovery Manager reads in a new log record. This is very inefficient and requires egregious amounts of memory access and copy. We can use caching techniques to reduce this performance overhead.

3.4.1 Metadata

Most of the metadata overhead is generated during the conversion step from a log record into a plan node. For each table, the DBMS needs to figure out its relevant metadata. These operations require frequent lookup into the database catalog. As long as those metadata are bound to some table id, the DBMS can cache those metadata and re-use them for the same table. Some significant metadata are listed below:

Physical to Logical Column Mapping

The Recovery Manager needs to maintain a mapping from physical column ids to logical column ids for each table. This is required because the column ids recorded in the log records are physical ids, while query compilation expects logical column ids. The Recovery Manager generates this mapping by accessing the database catalog.

Column Value Type Mapping

For redo records, the Recovery Manager also needs to maintain a mapping from each column to its corresponding SQL value type (e.g., **`INTEGER, FLOAT, VARLEN`**). The Recovery Manager uses this mapping to correctly copy the values out of each redo record into an insert/update plan node.

Catalog Access

Retrieving access to a database catalog is costly. Profiling results show that a single operation to retrieve catalog access can account for 10% of total instructions of a replay step. Therefore, the Recovery Manager uses a mapping to keep track of catalog accessors.

3.4.2 Compiled Query

A major benefit of query compilation is that the execution engine can reuse the compiled query. If the contents of a physical plan node remains largely unchanged, then there is no need to reconstruct the physical plan every time the Recovery Manager processes a log record.

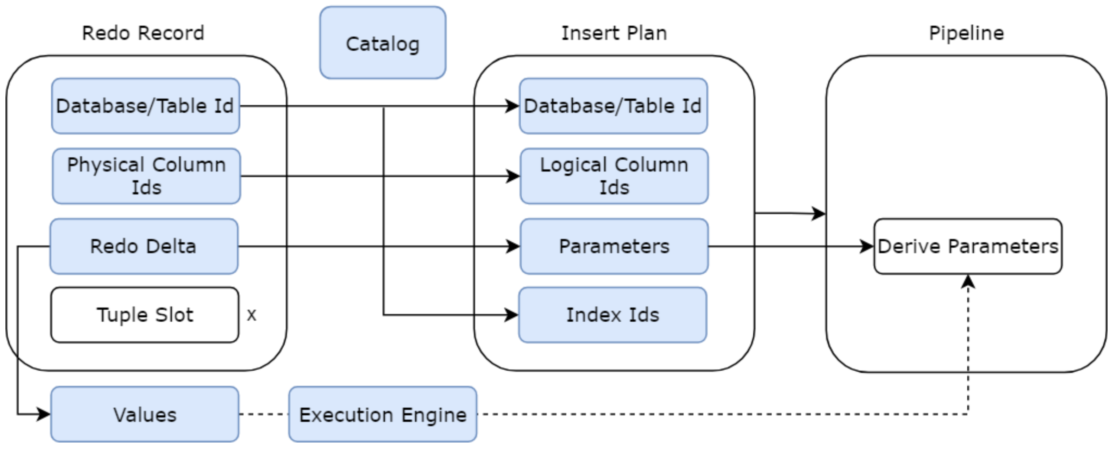


Figure 3.14: Parameterized insert plan.

We can cache each type of replay separately and use a combination of database and table id as unique identifiers. The Recovery Manager can re-use cached queries as long as the table schema associated with them remains unchanged.

Parameterization

Caching is trickier for replays that need to transfer values into plan nodes. We mentioned in Section 3.2.2 that insert/update needs to copy primitive values from the redo records into plan nodes. In most DBMSs, however, the plan nodes expect primitive values in some wrapper objects [16, 18] referred to as expressions. In NoisePage, the corresponding expression for primitive values is a constant value expression. As a result, the Recovery Manager needs to reconstruct a constant value expression for each new value.

A solution to this is to parameterize the constant value expression given to an insert/update plan node. We can represent an insert replay as **`INSERT INTO x VALUES @params`**. The Recovery Manager caches the plan nodes that contains parameter expressions. The parameters expressions only contain value type information, but do not contain any actual values. During run-time, the execution context passes in primitive values from the Recovery Manager into the plan node.

Chapter 4

Experimental Evaluation

We now evaluate the performance of our code generation recovery. The experiments focus on comparing the performance difference between code generation recovery and baseline recovery. Each experiment has two phases: the preparation phase and the recovery phase. During the preparation phase, we use three operations to generate log records:

- `INSERT` $\rightarrow$ **Insert Redo Record**: Create a random row with random values and insert into a random table.
- `UPDATE` $\rightarrow$ **Update Redo Record**: For a random row from a random table, update a random tuple slot with a random value.
- `DELETE` $\rightarrow$ **Delete Record**: For a random row from a random table, delete a random tuple slot.

We run these operations randomly for some number of times during the preparation phase. During the recovery phase, we load the log records into memory from disk and replay them with the Recovery Manager. We then measure the performance of only the replay step in our throughput measurement experiments.

We conduct all the experiments on a machine with the following specifications:

- **OS**: Ubuntu 20.04
- **CPU**: Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
- **RAM**: 188 GB 2 sockets, 20 cores/socket, 40 threads/socket

4.1 Throughput Measurement

Log replay generates transactions through commit records (See Section 3.1.2). Therefore, we can use transactions throughput to measure the performance of the recovery phase.

We use Google Benchmark for our measurement setup. The table starts with an initial table size of 10^6 tuples. During the preparation phase, we run five preparation operations for each transaction, with a total of 10^5 transactions. Each table uses a single column and **`INTEGER`** data type.

4.1.1 Insert

Statements Ratio	Baseline Throughput	Codegen Throughput
100% Insert	1.12×10^4 txns/s	6.00×10^3 txns/s

Table 4.1: Insert Replay Throughput.

We first measure the throughput (number of transactions executed per time period) of Insert Replays against the baseline recovery performance on a single table. The Recovery Manager uses only our insert replay implementation for recovery.

Functionality	IR%	Normalized IR%
Query Execution (Op Codes)	79.35	81.7%
Execution Context Initialization	9.35	9.62%
Retrieving Redo Values	6.19	6.37%
Metadata Cache Map Access	2.26	2.32 %
Compile	0.03	0.03 %

Table 4.2: Callgrind breakdown on an Insert Replay step. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

Operation Code	IR%	Normalized IR%	Description
`StorageInterfaceInit`	29.88	74.4%	Initializes a Storage Interface.
`StorageInterfaceTableInsert`	6.00	14.9%	Inserts a tuple into a table.
`StorageInterfaceFree`	2.76	6.87%	Deallocates a Storage Interface.
`StorageInterfaceGetTablePR`	1.40	3.48%	Retrieves a table's projected row.
Other	0.12	0.35%	

Table 4.3: Callgrind breakdown on the Query Execution step of an Insert Replay step.

Results show that the code generation throughput is 47% slower than the baseline implementation. Profiling result Table 4.11 explains that 81.7% of the slow down comes from the execution of the compiled query. This is unexpected, as we imangined that most of the overhead would come from memory copies for extracting values, but those memory copy overhead only accounts for 6%.

A further dive into the composition of query execution shows that 74.4% of the overhead comes from `StorageInterfaceInit`, which is a function that initializes a Storage Interface, as explained in Section 3.3.

4.1.2 Delete

We then measure throughput of Delete Replays against the baseline. The Recovery Manager uses only our delete replay implementation for recovery.

Statements Ratio	Baseline Throughput	Codegen Throughput
90% Insert, 10% Delete	1.12×10^4 txns/s	1.08×10^4 txns/s
75% Insert, 25% Delete	1.11×10^4 txns/s	1.07×10^4 txns/s
50% Insert, 50% Delete	1.09×10^4 txns/s	9.31×10^3 txns/s

Table 4.4: Delete Replay Throughput.

Code generation throughput results of Delete Replays are much closer to the baseline (90% to 95%), compared to those of Insert Replays. This is mainly because a delete operation does not require any memory copies from a redo record. Moreover, executing a delete operation is also a lightweight operation, as we have prevented the pipeline from performing extra work as explained in Section 3.2.2.

Functionality	IR%	Normalized IR%
Query Execution (Op Codes)	90.40%	91.46%
Execution Context Initialization	3.70%	3.74%
Delete Record Initialization	4.22%	4.27%
Metadata Cache Map Access	0.36%	0.36%
Compile	0.17%	0.17%

Table 4.5: Callgrind breakdown on a Delete Replay step.

Operation Code	IR%	Normalized IR%	Description
`StorageInterfaceInit`	24.35	76.27%	Initializes a Storage Interface.
`StorageInterfaceTableDelete`	5.80	18.16%	Deletes a tuple from a table.
`StorageInterfaceFree`	1.55	4.85%	Deallocates a Storage Interface.
Other	0.13	0.41%	
`StorageInterfaceGetTablePR`	0.10	0.31%	Retrieves a table's projected row.

Table 4.6: Callgrind breakdown on the Query Execution step of a Delete Replay step.

Callgrind analysis shows that, similar to an insert operation, a delete operation is heavily bottlenecked by query execution and initializing the Storage Interface.

4.1.3 Update

Finally, we measure throughput of Update Replays against the baseline. The Recovery Manager uses only our update replay implementation for recovery.

Statements Ratio	Baseline Throughput	Codegen Throughput
90% Insert, 10% Update	1.15×10^4 txns/s	7.46×10^3 txns/s
75% Insert, 25% Update	1.15×10^4 txns/s	7.31×10^3 txns/s
50% Insert, 50% Update	1.15×10^4 txns/s	6.68×10^3 txns/s

Table 4.7: Update Replay Throughput.

The performance of an Update Replay is similar to an Insert Replay. Code generated updates are more heavyweight compared to the baseline insert operations and update operations due to initialization overheads of the Storage Interface. Therefore, as the proportion of update statements increases, we notice a consistent drop on the code generation throughput.

Functionality	IR%	IR%/sum(IR%)
Query Execution (Op Codes)	80.08	83.50%
Execution Context Initialization	9.41	9.81%
Retrieving Record Values	4.98	5.19%
Metadata Cache Map Access	1.27	1.32 %
Compile	0.18	0.18 %

Table 4.8: Callgrind breakdown on an Update Replay step.

Operation Code	IR%	IR%/sum(IR%)	Description
`StorageInterfaceInit`	29.84	72.50%	Initializes a Storage Interface.
`StorageInterfaceTableUpdate`	8.42	20.46%	Updates a tuple of a table.
`StorageInterfaceFree`	2.78	6.75%	Deallocates a Storage Interface.
Other	0.12	0.29%	

Table 4.9: Callgrind breakdown on the Query Execution step of an Update Replay step.

Callgrind analysis shows that an update operation is also spending most of its time on query execution and creating the Storage Interface. Combined with results from Section 4.1.1 and 4.1.2, we can confirm that the allocation/deallocation Storage Interface becomes the main bottleneck of all the code generation replays.

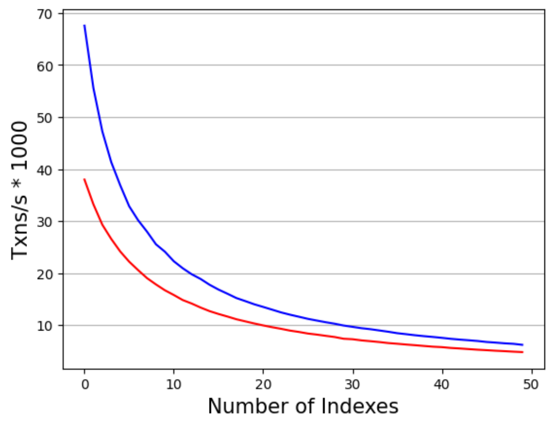
4.2 Throughput Measurement With Indexes

For in-memory DBMS, indexes are rarely stored on disk. Instead, they are reconstructed during log replay. However, this introduces design problems for an execution engine that has been adapted to work with compiled code. The recovery system needs to implement functionally redundant code to replay the indexes.

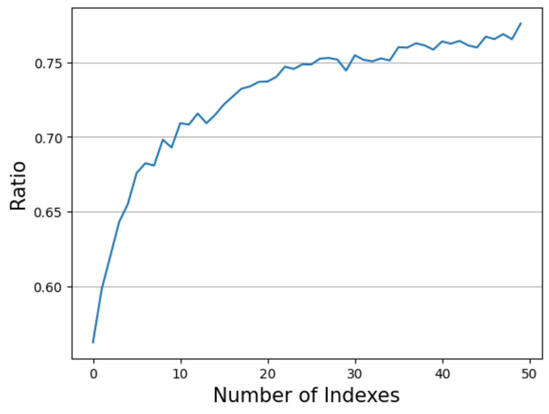
In the baseline recovery architecture, the Recovery Manager implements a custom index update function that is independent from the query execution engine. The code generation approach, however, uses the built-in execution engine for index updates. We believe as the number of indexes increases, the performance gap between baseline and the code generation approach will diminish.

For this experiment, we want to focus on the comparison for index recovery and avoid as much overhead from other operations as possible. Therefore, we go for a much lighter insert setup. However, it should be noted that some of the heavy overhead is still unavoidable (e.g., value copies, Storage Interface construction/destruction).

We create an empty table with a single column that accepts **`INTEGER`** data type. We execute 10^6 transactions to insert random integers into the table. For each data point, we create an increasing number of indexes for the table. Results show that as the number of indexes increases, the gap between the baseline and code generation implementation steadily decreases. Note that without any indexes, the gap between baseline and code generation is around two times, which matches the results from Table 4.1.



(a) Recovery throughput over the number of indexes. Red is baseline, blue is code generation.



(b) Throughput ratio between code generation and baseline implementation.

Figure 4.1: Recovery with Indexes.

Functionality	**IR%**	**Normalized IR%**
Index Update	96.23	96.3%
Insert	3.11	3.11%
Other (e.g., tuple map access)	0.57	0.59%

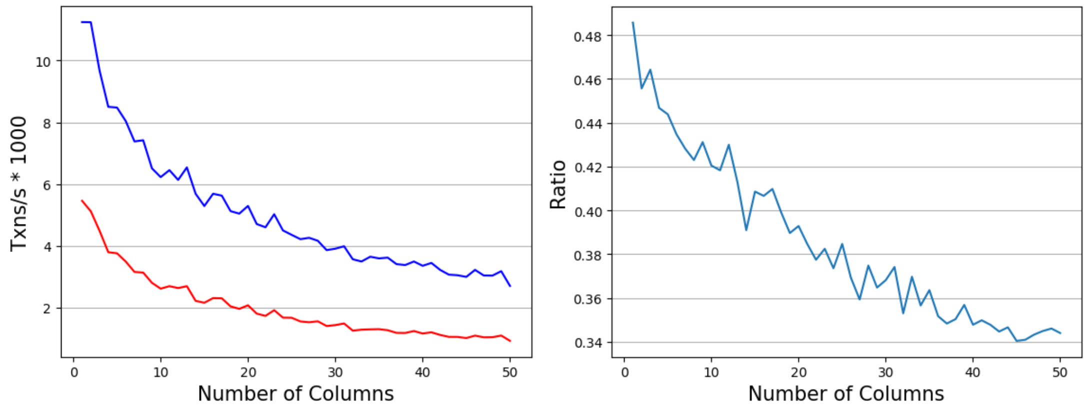
Table 4.10: Callgrind breakdown on Baseline Index Recovery. Number of indexes is 10.

Operation Code	**IR%**	**Normalized IR%**	**Description**
`StorageInterfaceIndexInsertUnique`	33.77	37.9%	Insert into an index.
`StorageInterfaceInit`	32.22	36.2%	Initializes a Storage Interface.
`StorageInertfaceGetIndexPR`	20.95	23.5%	Retrieves an index's projected row.
`StorageInterfaceTableInsert`	1.46	1.63%	Insert a tuple into a table.
`StorageInterfaceTablFree`	0.39	0.43%	Deallocates a Storage Interface.
`StorageInterfaceGetTablePR`	0.25	0.3%	Retrieves a table's projected row.
Other	0.02	0.02%	

Table 4.11: Callgrind breakdown on a Query Execution step of Code Generation Index Recovery. Number of indexes is 10. The query execution step accounts for 96% IR for a single replay step.

A deeper dive into the implementations shows a percentage difference in index updates between the two approaches. Index updates account for 96.3% of total instructions in the baseline implementation. On the other hand, for the code generation approach, index update accounts for $37.9\% + 23.5\% = 61.4\%$ of total instructions. This implies that the code generation requires less overhead to prepare for an insert index operation.

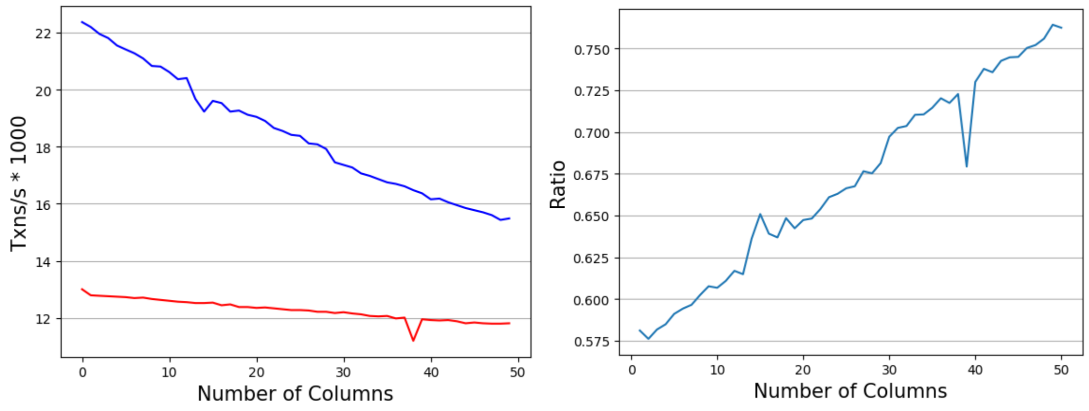
4.3 Throughput Measurement With Scaling Columns



(a) Throughput over the number of columns. Red is baseline, blue is code generation.

(b) Throughput ratio between code generation and baseline implementation.

Figure 4.2: Recovery with Scaling Number of Columns.



(a) Throughput over the number of columns. Red is baseline, blue is code generation.

(b) Throughput ratio between code generation and baseline implementation.

Figure 4.3: Recovery with Scaling Number of Columns. Number of Indexes is 10.

Chapter 5

Related Work

Our code generation approach applies for DBMSs that use physical logging for recovery. It does not apply to DBMSs that use other logging schemes, such as logical logging and physiological logging. For instance, VoltDB[22] uses a form of physiological logging known as command logging. The DBMS records the transactions instead of physical changes in the log records.

Our method also does not extend to systems that do not have logging architectures. SAP HANA[21] uses a log-less approach through parallel table replication.

Chapter 6

Conclusion and Future Work

We presented a code generation recovery approach for DBMS. Our method integrates the code generation Execution Engine with the recovery system by converting log records into compiled native code. We provided an overview of the logging and query compilation architecture of NoisePage. We then showed how we implemented our approach within NoisePage. Finally, we evaluated our approach with throughtput experiments to show that while code generation recovery has worse recovery throughput compared to NoisePage baseline recovery, it offers less engineering overhead and is more efficient in processing index updates.

Our results show that the initialization process for executing compiled queries (i.e., creating storage interface) is the main bottleneck for code generation recovery in NoisePage. One possible optimization is to cache reusable objects that are repeatedly initialization during query execution inside the recovery system or the execution engine.

Bibliography

[1] Rakesh Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. IWDM'89, pages 269–285. 2.1

[2] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. Constant time recovery in azure sql database. *Proc. VLDB Endow.*, 12(12):2143–2154, August 2019. ISSN 2150-8097. URL https://doi.org/10.14778/3352063.3352131. 2.1

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. SIGMOD '95, page 1–10, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917316. URL https://doi.org/10.1145/223784.223785. 3.1.1

[4] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005. 1

[5] CMUDB Group. The noisepage system. https://noise.page/, June 2020. 1, 1.1, 2.1, 3

[6] S.S. Conn. Oltp and olap data integration: a review of feasible implementation methods and architectures for real time data analysis. In *Proceedings. IEEE SoutheastCon, 2005.*, pages 515–520, 2005. doi: 10.1109/SECON.2005.1423297. 1

[7] Transaction Processing Performance Council. Tpc benchmark status, 2015. 3.1.1

[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Paul Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *ACM International Conference on Management of Data 2013*, June 2013. URL https://www.microsoft.com/en-us/research/publication/hekaton-sql-servers-memory-optimized-oltp-engine/. 1, 2.1

[9] MySQL Documentation. Web, 2021. 2.1

[10] PostgreSQL Documentation. Web, 2021. 2.1

[11] Frans Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Tjomas Neumann, and Andrew Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8: 1–130, 01 2017. doi: 10.1561/1900000058. 1, 2.1

[12] Michael Franklin. Concurrency Control and Recovery. *The Computer Science and Engineering Handbook*, pages 1058–1077, 1997. 1, 2.1

[13] Craig S. Freedman, Erik Ismert, and P. Larson. Compilation in the microsoft sql server

hekaton engine. *IEEE Data Eng. Bull.*, 37:22–30, 2014. 1, 2.2

[14] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992. doi: 10.1109/69.180602. 1

[15] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet Wiener. Fast database restarts at facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 541–549, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2595642. URL https://doi.org/10.1145/2588555.2595642. 1, 2.1

[16] G. Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. doi: 10.1109/69.273032. 3.4.2

[17] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. URL https://doi.org/10.1145/289.291. 1

[18] T. Kersten, Viktor Leis, A. Kemper, Thomas Neumann, Andrew Pavlo, and P. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11:2209–2222, 2018. 1, 3.4.2

[19] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014. 2.2

[20] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 613–624. IEEE, 2010. 2.2

[21] Juchang Lee, Wook-Shin Han, Hyoung Jun Na, Chang Gyoo Park, Kyu Hwan Kim, Deok Hoe Kim, Joo Yeon Lee, Sang Kyun Cha, and Seunghyun Moon. Parallel replication across formats for scaling out mixed oltp/olap workloads in main-memory databases. *The VLDB Journal*, 27(3):421–444, June 2018. URL https://doi.org/10.1007/s00778-018-0503-z. 5

[22] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615, 2014. 1, 2.1, 5

[23] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11:1–13, September 2017. 1, 2.2

[24] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992. ISSN 0362-5915. doi: 10.1145/128765.128770. URL https://doi.org/10.1145/128765.128770. 1, 2.1, 3.1.2

[25] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011. 1, 2.2, 3.2.1

[26] Drew Paroski. Code Generation: The Inner Sanctum of Database Performance. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html, September 2016. 1, 2.2

[27] D. P. Reed. Naming and synchronization in a decentralized computer system. 1978. URL https://dspace.mit.edu/handle/1721.1/16279. 3.1.1

[28] Caetano Sauer, Goetz Graefe, and Theo Härder. Instant restore after a media failure. *CoRR*, abs/1702.08042, 2017. URL http://arxiv.org/abs/1702.08042. 1

[29] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 465–477, USA, 2014. USENIX Association. ISBN 9781931971164. 1, 2.1