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

Tianlei Pan

CMU-CS-21-131

June

Computer Science Department
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

I would like to thank my professor Andy Pavlo for guiding me through my journey at the CMU Database Group. His passion, energy and commitment motivated me to pursue research in database systems in the first place. I will carry on with my life with the valuable lessons that I learned from him. I would also like to thank my peers in the CMU Database Group for working with me and carrying me through my research work. Finally, I want to thank my family for their unwavering support.

I wish you all the best.

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[21]. 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 [32]. 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[21]. 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 [39] 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[29]. The ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) [32] protocol is a recovery method developed by IBM in the 1990s. It guarantees database integrity in the fact of Transaction, System and Media Failures [32]. 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) [13].

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 [16]. 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, 7, 17, 29, 41]. 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 [12]. An example is the recovery of database indexes [12]. Disk-based DBMSs log updates to index structures (e.g., B+ Tree) that allow faster recovery during log replay [32]. In comparison, in-memory DBMSs do not log index updates [5, 7, 29, 41] and choose to reconstruct indexes from scratch during log replay. Despite the differences, many in-memory DBMSs still use some form of logging [5, 7, 29, 41] in their recovery systems.

In addition to changes in recovery systems, in-memory DBMSs adopt various optimization algorithms for their execution engines[22]. 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[22]. During execution, the DBMS breaks down a query into various tasks [5, 33]. 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 [33]. Moreover, the compiled code can be optimized around locality that increases the chance of a data tuple being propagated between operators in CPU registers[30].

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[22]. The DBMS also generates low-level machine code that is hard to understand and debug [22].

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, 14, 33, 35] 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 pro-

cess 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 potentially more efficient in index updates. 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** [13]: None of the changes made by aborted or failed transactions are persisted visible after recovery.

Many DBMSs [2, 5, 7, 8, 9, 29, 41] 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 [32], 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 [32]. 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, 7, 17, 29, 41]. 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, 7, 17, 29, 41]. 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 [12]. During recovery, the DBMS reconstructs

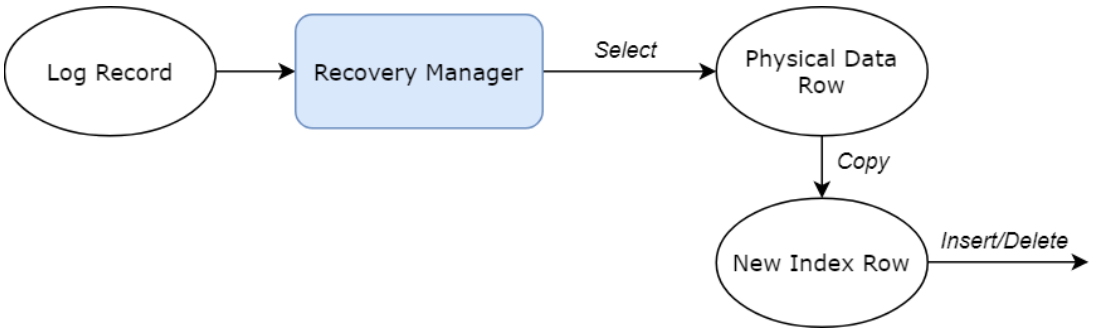


Figure 2.1: Index Update During Recovery.

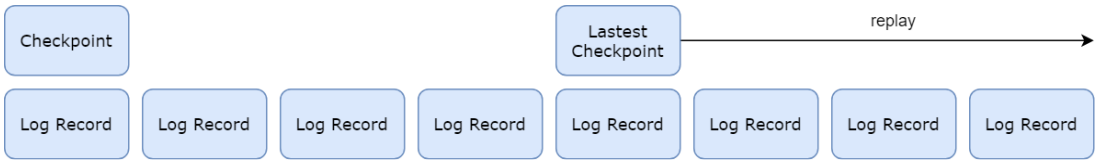


Figure 2.2: Recovery with Log Records and Checkpoints.

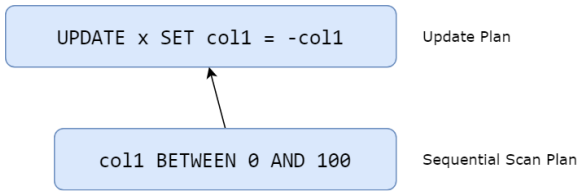


Figure 2.3: Physical Plan that corresponds to **`UPDATE x SET col1 = -col1 WHERE col1 BETWEEN 0 AND 100`**.

the database with redo records reconstructs and create the indexes simultaneously [7, 29]. 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

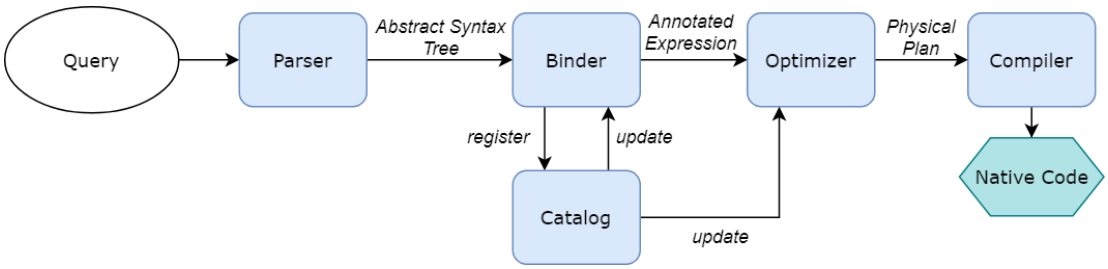


Figure 2.4: Query Compilation Architecture.

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) [23, 25]. This approach is used in Amazon Redshift and pre-2016 MemSQL [35]. 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 [30], Hyper [33], Hekaton [14], and SingleStore [35].

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

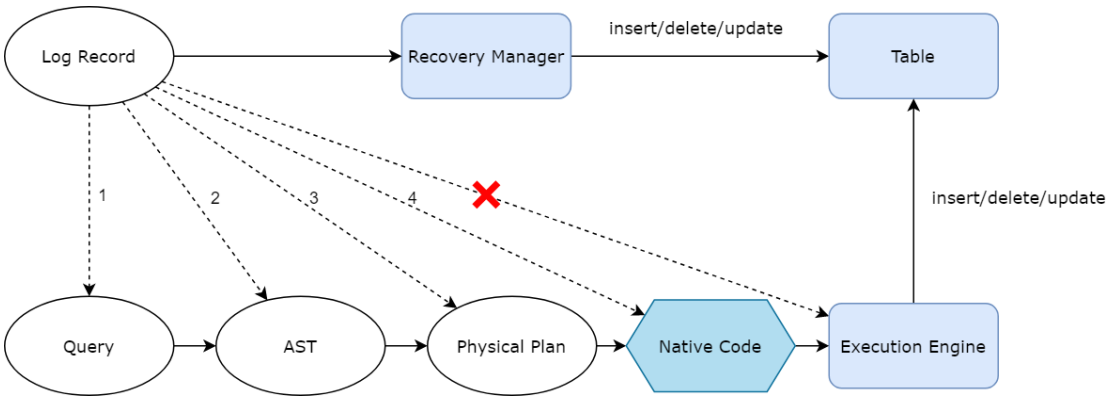


Figure 2.5: Possible ways to integrate a physical logging system with code generation.

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 (Figure 2.1).

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. Figure 2.1) 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. For instance, if an index contains expressions (e.g., **`CREATE INDEX c ON A (cola + colb)`**), then the Recovery System needs to implement logic to evaluate those expressions as well.

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 NoisePage's recovery system and query compilation pipeline to support log replay with code generation. Our implementation follows 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 extend 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 and compile them into native code (Recovery Code Generation).
2. Run the compiled machine code using the execution engine (Recovery Execution) for log replay.

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 NoisePage's 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's Transaction Manager uses a multi-versioned delta store [37] that ensures snapshot isolation [3]. The Transaction Manager allows simultaneous read-write, but does not

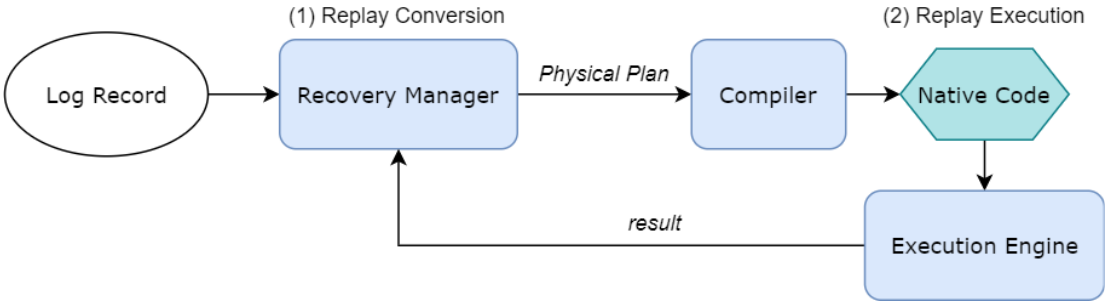


Figure 3.1: Code Generation Recovery Mechanism.

allow write-write conflicts on a per tuple basis. A transaction's lifecycle 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. A tuple slot stores the offset of a tuple in a data block. It is a combination of: (1) physical memory address of the block with the tuple, and (2) its offset logical within the block [27]. 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 copies of the tuple, but stores delta information (redo records) about the updates instead [40]. 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 [41]. 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 one 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 the Recovery Manager to replay corresponding records. 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 the DBMS to write some log records from an aborted transaction to disk. During recovery, the DBMS needs to remove aborted transactions.

3.1.2 Log Manager

The Log Manager (Figure 3.2) is responsible for performing write-ahead logging [32] in NoisePage. Each log record is self-contained and does not require any additional metadata to be replayed. As we now describe, the Log Manager consists of two separate tasks that run in different threads: (1) Log Serializer and (2) Log Consumer.

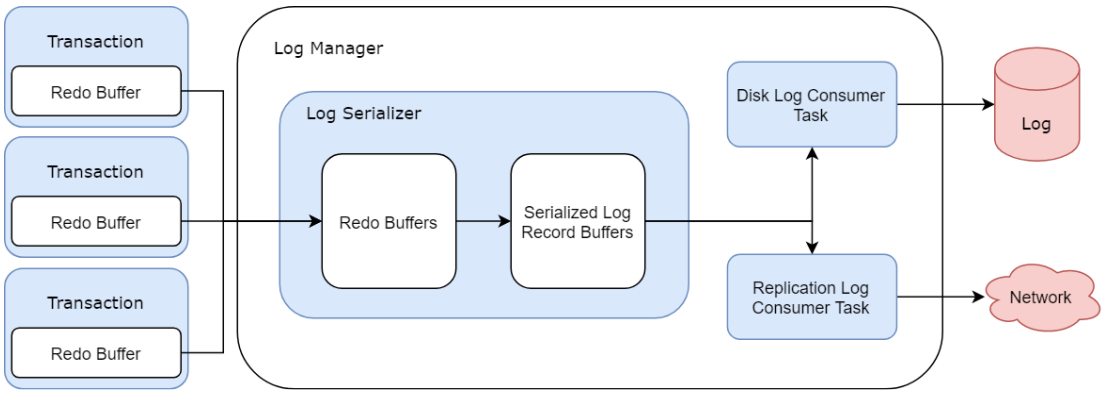


Figure 3.2: NoisePage Logging.

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 splits 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 generated by an individual transaction is guaranteed to appear in the same order as they were created. This property helps ensure the snapshot isolation property of NoisePage during log replay.

Log Consumer

Multiple Log Consumers are responsible for consuming buffers supplied by the Log Serializer. The Log Serializer provides each Log Consumer a copy of the log record 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.

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. The Recovery Manager loads the log records into memory and use them to performs 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 redo and delete records inside a deferred changes buffer [36], ordered by their transaction

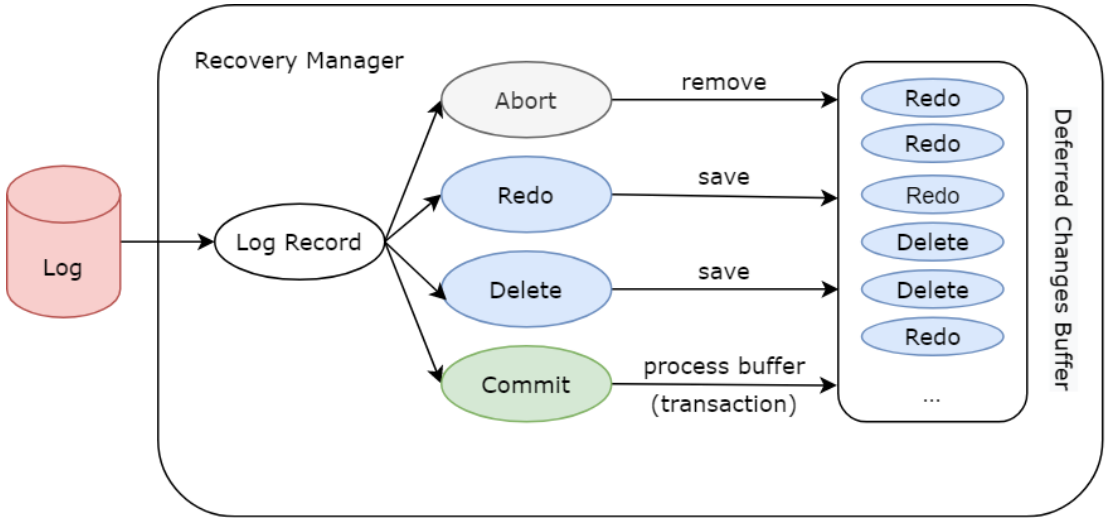


Figure 3.3: Recovery Manager.

(a) Redo Record Structure.

(b) Delete Record Structure.

Figure 3.4: Log Record.

timestamp. An abort record removes a corresponding redo/delete record from the buffer using the begin transaction timestamp (Figure 3.4). 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 (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 be 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. The recovery manager will retrieve the new tuple slots after new values are inserted into tables. It then maps the old tuple slots to the new tuple slots. The tuple slot mapping allows the Recovery Manager to apply changes to the correct memory locations in the new memory environment. For instance, when the Recovery Manager reads in a delete record, it finds the

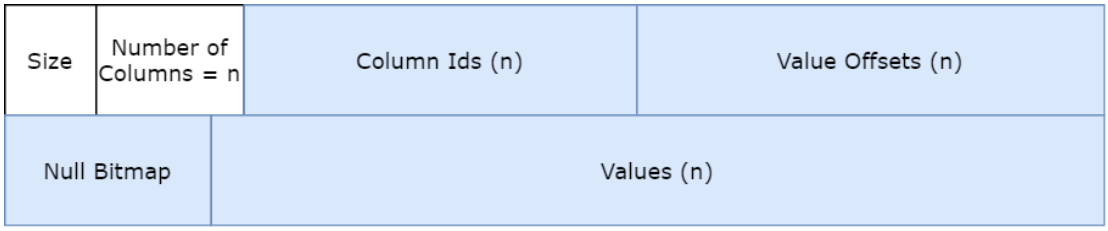


Figure 3.5: Redo Record Projected Row.

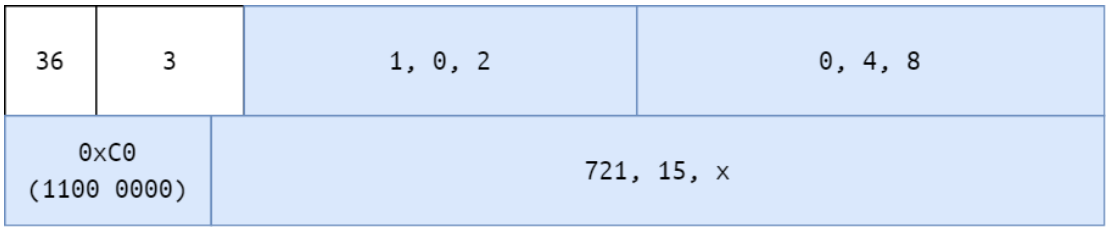


Figure 3.6: Projected Row that represents [15, 721, **`NULL`**].

new tuple slot within the mapping with the original tuple slot contained in the log record and performs a delete on the new tuple slot.

While storing tuple slots is sufficient for delete records, redo records require additional data 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 row in a database table. The DBMS can access an individual value within the projected row by identifying a column id and reading the value using that column's value offset. For instance (Figure 3.6), with column ids `[1,0,2]`, the DBMS can find their values with the value offset array `[0,4,8]`. In the null bitmap, it shows that only the first two columns are non-null. This points to the values `[721, 15, NULL]`. And if we reorder them according the column ids, we get row `[15, 721, NULL]`.

Replay Step

The Recovery Manager processes each log record differently depending on its record type. Commit and abort records do not change the physical storage of the the database. Therefore, to simplify, we only need to consider the effects of redo and delete records.

We refer to a replay step as 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 Re-

covery 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. While NoisePage does not use checkpoints [32] for recovery, we want to point out that this method will not work if the DBMS loads a checkpoint before log replay. The Recovery Manager has no means of storing the tuple slots created during checkpoint restoration.

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 on how NoisePage employs code generation for query execution. We then show how the Recovery Manager converts 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 DBMS interprets a query, parses it into an AST and converts it into a physical plan tree. A plan tree consists of plan nodes and specifies how the DBMS should execute the query on the physical level of the database.

Code generation is responsible for compiling physical plans into machine code. The code generation pipeline in NoisePage follows from the data-centric compilation approach[33]. The DBMS feeds each plan node into a matching plan translator. Each plan translator correspond to a plan node in the original plan tree. For instance, an insert plan node has a matching insert plan translator; a delete plan node has a matching delete plan translator. A plan translator converts a physical plan tree/node into an operator pipeline. The operator pipeline contains an ordered collection of relational operators (e.g. insert, get table row), ended with a pipeline breaker (e.g., sort tuples, creating a hash table). The execution engine only materializes the pipeline results when it encounters a pipeline breaker.

NoisePage then compiles each operator pipeline into a customized imperative language (TPL) [24, 30]. The DBMS then compiles each function of a query inside TPL into bytecode (TBC). The DBMS can choose to interpret the bytecode with a built-in virtual machine, or further compile it 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 to delete. It then performs a delete operation on those 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 executed in the DBMS. After this, the DBMS compiles the operator pipeline into machine code. Each function inside the machine code matches to an operation code function inside the execution engine. When the execution engine runs the compiled code, it invokes its corresponding operation code functions. For

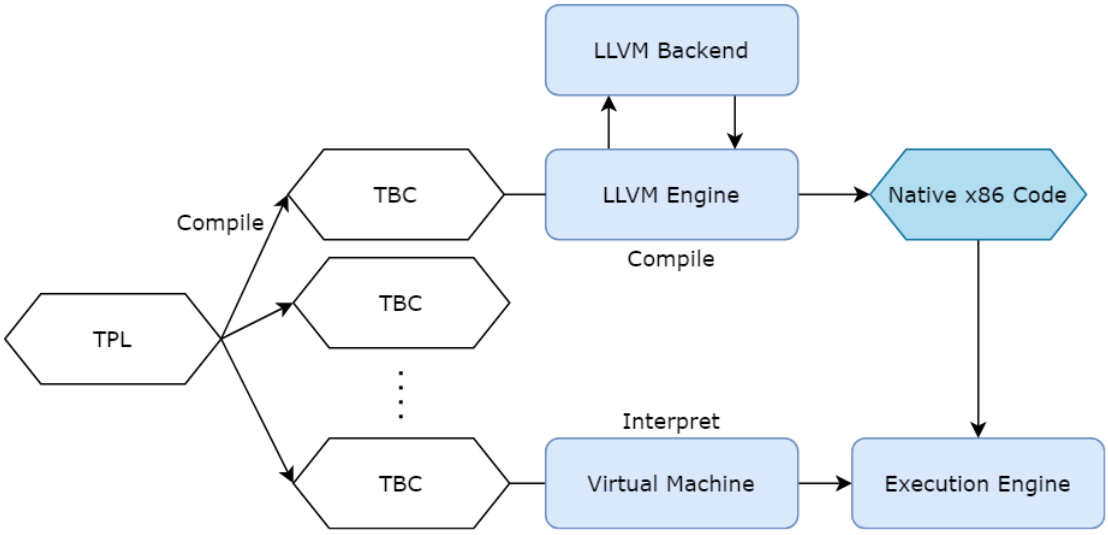


Figure 3.7: Lifecycle of a TPL code fragment. Execution can run in either interpret or compile mode.

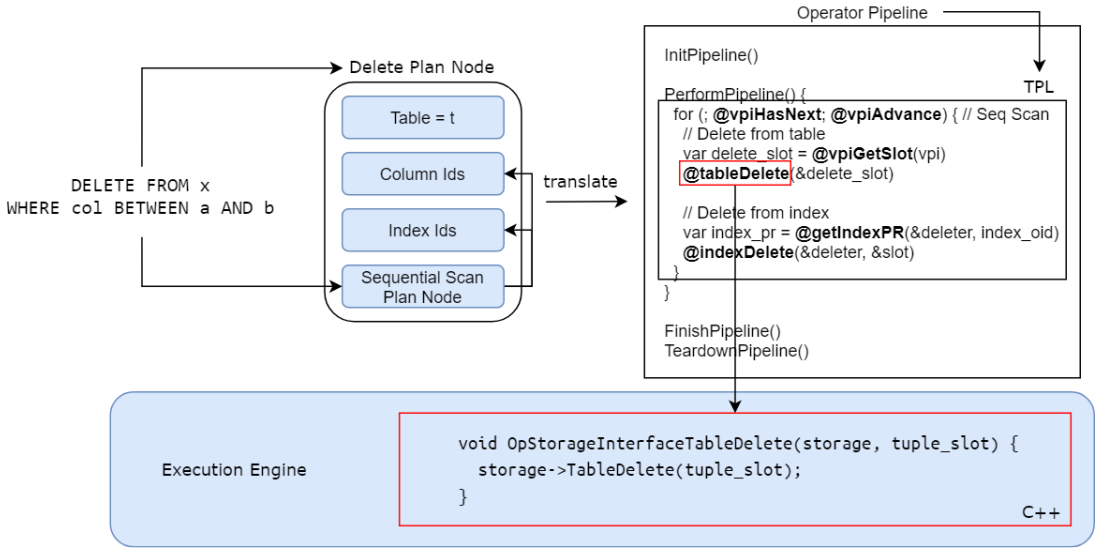


Figure 3.8: Code Generation of **`DELETE FROM x WHERE col BETWEEN a AND b`**

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 Replay Conversion

We consider each type of replay as a form of SQL statement that the DBMS convert into native code through the code generation process discussed above, as shown in Figure 3.9. We further divide the code generation process for a log record into three steps: (1) plan generation (creating the physical plan node), (2) plan translation (translating the plan into operator pipelines), and

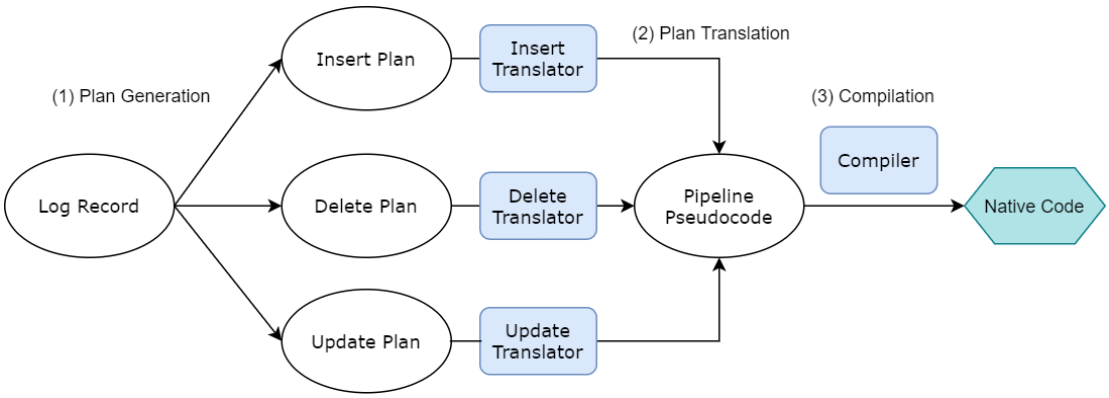


Figure 3.9: Process for converting log records into native code.

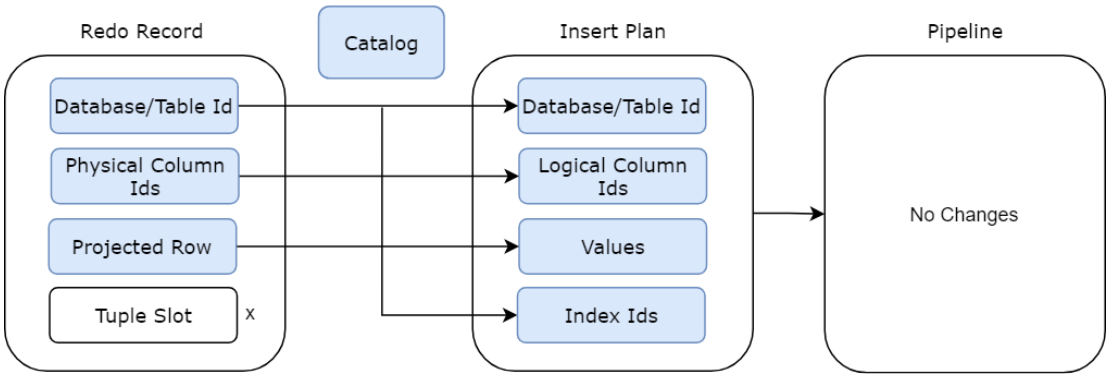


Figure 3.10: Conversion from an Insert Redo Record into an Insert Plan Node. The tuple slot provided by the redo record is discarded.

(3) compilation.

In plan generation, the Recovery Manager converts each replay process (insert/update/delete replay) into a corresponding plan node. In plan translation, the Recovery Manager uses matching plan translators to convert physical plans into operator pipelines. 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 retrieves 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). A redo record stores those values as constants (e.g., integer, float, double). The Recovery Manager then feeds those constants into the insert plan node. This conversion requires no changes to the plan nodes, nor to the operator pipelines.

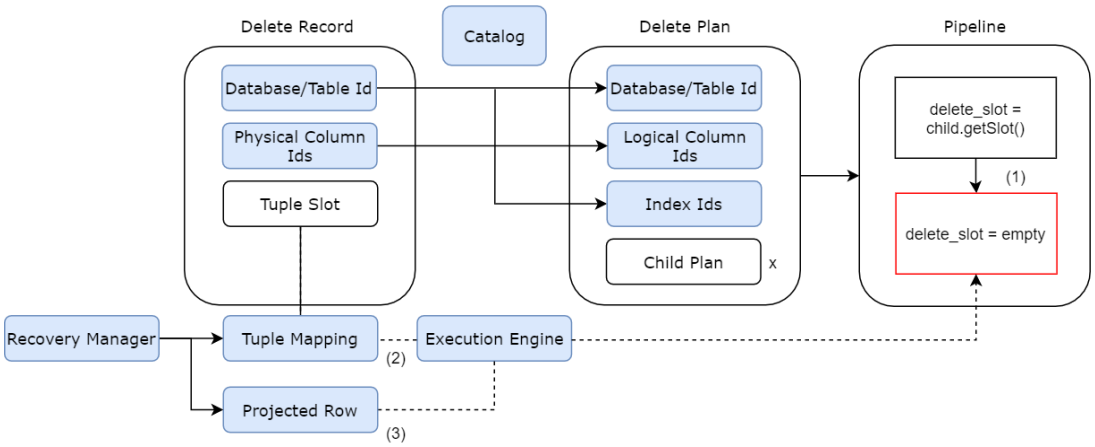


Figure 3.11: Conversion from a Delete Redo Record into a Delete Plan Node. The operator pipeline uses the tuple slot from the delete record.

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), unless a **`DELETE`** statement wants to delete everything from a table. 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 delete.

However, a delete record's tuple slot, combined with the tuple mapping from the Recovery Manager, already points to the tuple slot to delete. In this case, there is no need for the plan node to go through the child node and resolve the tuple slot for deletion. Moreover, delete records do not contain values like a redo record does (Figure 3.4). The delete plan needs to use those values to delete them from the indexes. Therefore, we need to provide the operator pipeline with a projected row that contains values that correspond to the row of the tuple slot from the Recovery Manager.

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

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 or expression) that are expensive to construct. 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 from redo records.

We therefore perform the following changes (Figure 3.12): (1) extend the update plan node

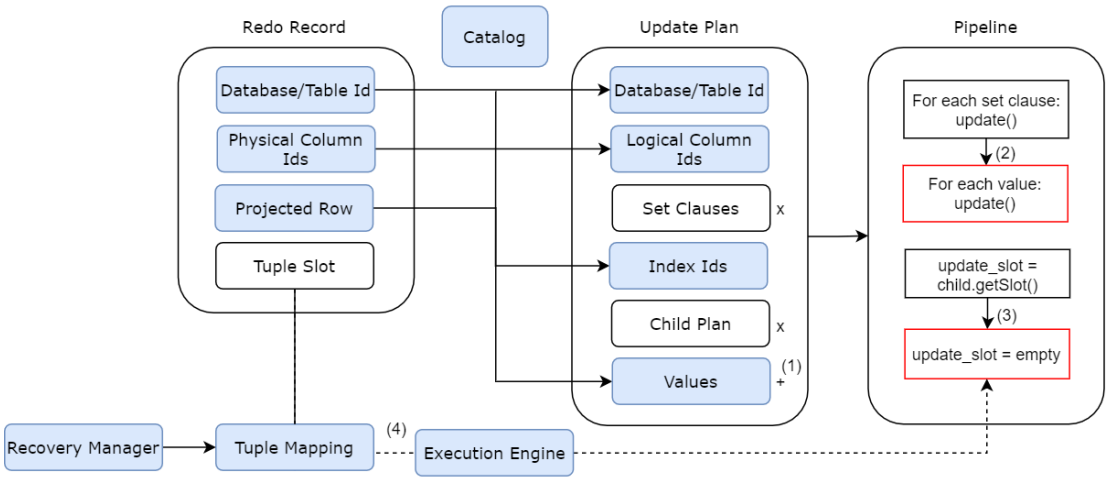


Figure 3.12: Conversion from an Update Redo Record into an Update Plan Node. **SET** clauses and child plan nodes inside the update plan node are discarded.

to accept primitive values like an insert plan node does, (2) modify the operator pipeline to process the constant 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 database objects (e.g., tuple slots, projected row) from the Recovery Manager. We modify the execution engine to achieve this. 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 Replay Execution

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 preparation 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 to execute, database memory pool). We pass the log record from the Recovery Manager into the execution context.

During recovery, we modify functions within the execution engine so that they will use the objects provided by the Recovery Manager through the execution context (Figure 3.13).

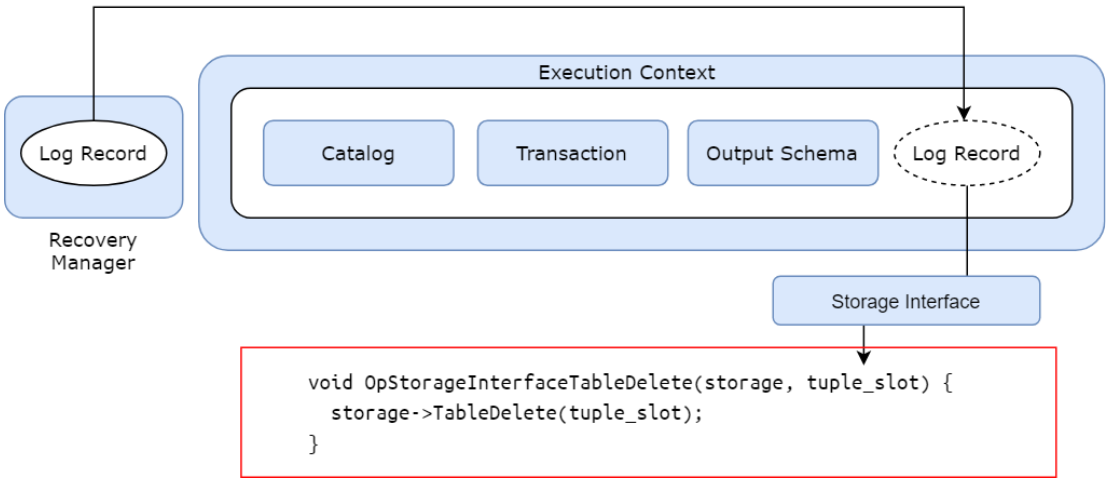


Figure 3.13: Execution engine functions can access objects from the Recovery Manager through an Execution Context.

3.4 Caching Optimization

Our 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 search overhead is from the replay conversion step. For each table, the DBMS needs to figure out its relevant metadata. These operations require frequent lookup into the database catalog. However, all of those metadata are bound to a fixed database and table id. Therefore, the DBMS can cache those metadata with a hash table and re-use them for the same table.

Physical to Logical Column Mapping

The Recovery Manager maintains 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. On the other hand, query compilation expects logical column ids because the execution engine is designed to work with upper layers of a DBMS. The Recovery Manager generates this mapping by accessing the database catalog.

Column Value Type Mapping

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

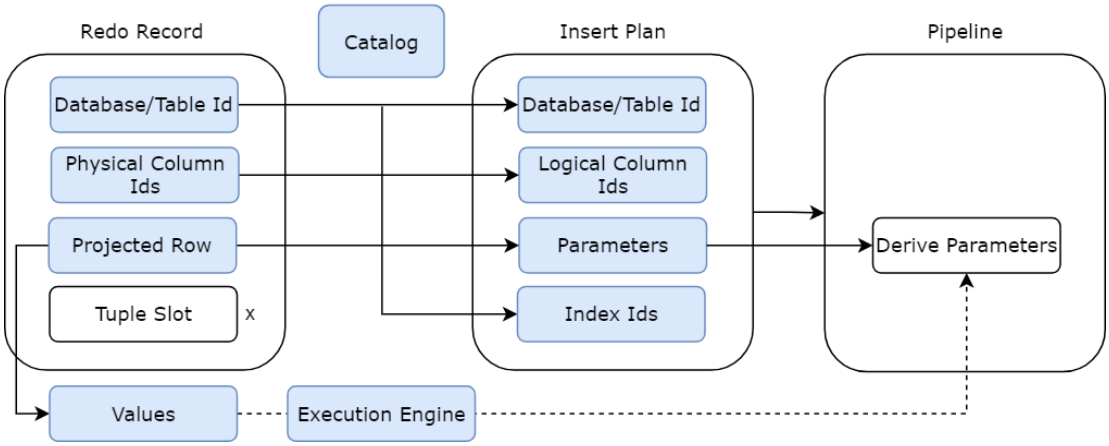


Figure 3.14: Parameterized insert plan.

Catalog Access

It is costly for the Recovery Manager to repeatedly read from the database catalog. 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 hash table to keep track of catalog readers.

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 unchanged, then there is no need to reconstruct the physical plan every time the Recovery Manager processes a log record. For instance, we only need to create a new delete plan node if we receive a delete record with a new database and table id.

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. However, the query cache needs to be updated if the table schema associated with the compiled query is modified. For instance, if a table is dropped, the query cache should remove all queries related to that table from the cache.

Parameterization

Caching is trickier for replay types 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 some DBMSs, however, the plan nodes expect primitive values in some wrapper objects [18, 22] 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 ?`. The Recovery Man-

ager 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 our code generation recovery method 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` → **Insert Redo Record**: Create a random row with random values and insert into a random table.
- `UPDATE` → **Update Redo Record**: For a random row from a random table, update a random tuple slot with a random value.
- `DELETE` → **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 on demand 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
  2 sockets, 20 cores/socket, 40 threads/socket
- **RAM**: 188 GB

We use Google Benchmark for our test setup. Google Benchmark is a C++ library that provides code snippets similar to unit tests [28]. We implement our experiments using the Google Benchmark framework to count the number of transactions per second a DBMS can execute. We enable data collection only when recovery starts.

We use Callgrind for our test result analysis. Callgrind is a profiling tool that records function calls history in a program [11]. The collected data consists of the number of instructions executed per function, the functions' relationship to source code, and caller/callee relationship between functions. We use Callgrind on the NoisePage binary to record this data. We then use the profile data to analyze the instructions overhead breakdown within the log replay process.

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. Baseline recovery executes 53% more transactions per second than code generation recovery.

Functionality	IR%	Normalized IR%	Description
Query Execution	79.35	81.7	Executes the insert replay query.
Execution Context Initialization	9.35	9.62	Initializes an execution context (Figure 3.13) for query execution.
Value Copies	6.19	6.37	Copy values from a redo record into a physical plan.
Cache Access	2.26	2.32	Reading and writing values/queries into hash table caches.
Compile	0.03	0.03	Compiling the insert replay query.

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.

Function	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. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

4.1 Table Recovery

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

The table starts with an initial table size of 10^6 tuples. During the preparation phase, we run five insert/update/delete statements for each transaction, with a total of 10^5 transactions. Each table uses a single column that is **`INTEGER`** data type.

4.1.1 Insert

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 code generation insert replay implementation for recovery.

Results show that the code generation throughput is 47% slower than the baseline implementation. To investigate this, we run the NoisePage binary with the Callgrind tool and only enable data collection during recovery. We collect the instructions read percentage (IR%) for different function calls that occurred during recovery. To facilitate comparison, we normalized those percentages to reflect each function's IR overhead relative to others. Table 4.2 shows a breakdown of the instructions read percentage reported by Callgrind.

- **Query Execution** refers to the process of the execution engine running the compiled replay query. This is where the DBMS performs the a recovery replay step and should take up most of the overhead.
- **Execution Context Initialization** is a preparation phase for query execution. In NoisePage, it creates an execution context (Figure 3.13) that is necessary for execution. 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 to execute, database memory pool).
- **Value Copies** refers to the step where the Recovery Manager copies values from redo records into an insert/update physical plans. Value copies can potentially take up a lot of overhead as the number of columns of a table increases.
- **Cache Access** occurs whenever the Recovery Manager tries perform reads/writes on a metadata/compiled query hash table.
- **Compile** step refers to the compilation of an insert/delete/update replay query. Compilation should take little time, since most of the queries reside in the cache.

The profiling result in Table 4.2 explains that 81.7% of the slow down comes from the query execution step of the compiled query. This is expected, as the compiled query represents an insert replay. Therefore, the execution of the query should take up most of the time during an insert replay step. Execution context initialization and value copies together also account for 16% of the total overhead. This overhead is unavoidable, since the DBMS needs the execution context to execute the query and the values to construct physical plan nodes.

The composition of query execution shows that 74.4% of the overhead comes from `StorageInterfaceInit` (Table 4.3), which is a function that initializes a storage interface. The storage interface object points to a database table and provides access to its columns, projected rows, and indexes. It allows the execution engine to manipulate over a database table to perform an insert, delete or update. In NoisePage, every time the execution engine begins executing an insert/delete/update pipeline, it first creates the storage interface object. The construction of this object proves to be expensive. Therefore, the main overhead of insert code generation recovery comes from execution initialization. We also believe that the percentage of this overhead should remain similar for the next two experiments, since the initialization of the storage interface does not depend on the type of operation that the DBMS is executing.

We further study the performance difference between the baseline insert replay and code generation insert replay by inspecting their implementations. To perform an insert replay in baseline recovery, the Recovery Manager only needs to (1) initialize a new redo record for insert by copying from the original redo record, (2) insert the new redo record into the table, and (3) update indexes. In step (1), baseline recovery only needs to copy the entire redo record.

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 recovery throughput falls behind baseline recovery throughput by 10% on average.

It uses this copy directly as the log record for this replay step. Meanwhile, code generation recovery needs to iterate over each value inside the redo record, cast them to the correct type, and copy the casted values into a new array. These operations result in the additional value copies overhead. Baseline recovery also does not need any execution initialization and cache access that are specific to code generation recovery. In step (2), baseline recovery is performing the same operation as `StorageInterfaceTableInsert`. While for code generation recovery, it performs additional work aside from `StorageInterfaceTableInsert` to construct and destruct the storage interface. This accounts for 89.3% of the total IR overhead and proves to be the main bottleneck for code generation recovery. For step (3), the tables used in the experiments do not contain indexes. We will verify the effects of indexes in later sections.

4.1.2 Delete

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

Code generation throughput results of Delete Replays are much closer to the baseline (90% to 95%), compared to those of Insert Replays. A delete operation does not require value copies from a redo record. This avoids the value copy overhead that is persistent in insert replay recovery.

Moreover, comparing to an insert operation, a delete operation is also more lightweight. For an insert replay, its operator pipeline needs to loop over all the values and resolve the tuple slot for insertion. As shown in Section 3.2.2, we removed the step for a delete operator pipeline to look for a new tuple slot and allows it to retrieve the tuple slot directly during execution. A delete operator pipeline also does not need to loop over any values, since it directly deletes the tuple slot provided during execution.

The Callgrind results in Tables 4.5 and 4.6 show that, similar to an insert operation, a delete operation is heavily bottlenecked by execution initialization (`StorageInterfaceInit`). Both insert and delete replays have similar IR% (74% vs 76%) of execution initialiation overhead. On the other hand, the delete operation (`StorageInterfaceTableDelete`) itself accounts for only 9.81% of the total overhead. This matches our expectations. As we explained earlier, delete replays are more lightweight than insert replays. An insert (StorageInterfaceTableInsert) costs 18.16% overhead, which is larger than the overhead (9.81%) from a delete (`StorageInterface-TableDelete`).

We further examine the differences between baseline delete replay and code generation

Functionality	IR%	Normalized IR%	Description
Query Execution	90.40%	91.46%	Executes the delete replay query.
Execution Context Initialization	3.70%	3.74%	Initializes an execution context (Figure 3.13) for query execution.
Delete Record Initialization	4.22%	4.27%	Initializes a new delete record to be used during execution.
Cache Access	0.36%	0.36%	Reading and writing values/queries into hash table caches.
Compile	0.17%	0.17%	Compiling the delete replay query.

Table 4.5: Callgrind breakdown on a Delete 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`	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. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

delete replay to understand the drop in recovery throughput. To perform a delete replay in baseline recovery, the Recovery Manager needs to: (1) use the tuple slot in the delete record to generate a new delete record, (2) collect the table's column ids and fetch the values that reside in the tuple slot to perpare for index updates, (3) delete from the table, and (4) delete from the indexes. Code generation recovery does step (1) and (2) in delete record initialization, which only takes up 4.27% IR%. The main performance difference comes in step (3). In baseline recovery, step (3) is equivalent to executing `StorageInterfaceTableDelete`. In code generation recovery, it needs to do additional work to create and free the storage interface object for each delete replay.

4.1.3 Update

Finally, we measure throughput of Update Replays against the baseline. The Recovery Manager uses only code generation update replay implementation for recovery. The performance of an Update Replay is similar to an Insert Replay, as they both require copying values into their physical plans. Therefore, due to execution initialization overhead, code generated updates are more heavyweight compared to the baseline insert and update operations.

The throughput measurement results demonstrate the effect of execution initialization overhead. As the proportion of update statements increases, we notice a consistent drop in the code

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. Code generation recovery throughput falls behind baseline recovery (from 64.9% to 58.1%) as the percentage of update statements increases.

Functionality	IR%	Normalized IR%	Description
Query Execution	80.08	83.50%	Executes the update replay query.
Execution Context Initialization	9.41	9.81%	Initializes an execution context (Figure 3.13) for query execution.
Retrieving Record Values	4.98	5.19%	Copy values from a redo record into a physical plan.
Cache Access	1.27	1.32 %	Reading and writing values/queries into hash table caches.
Compile	0.18	0.18 %	Compiling the update replay query.

Table 4.8: Callgrind breakdown on an Update Replay step. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

Function	IR%	Normalized 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. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

generation throughput. When the preparation statements consist of 10% updates, code generation recovery throughput is 64.9% of baseline recovery throughput. As the percentage of update statements increases, baseline recovery throughput remains the same. However, code generation recovery throughput is only 58.1% of baseline recovery throughput when the percentage of update statements reaches 50%. This is because as the percentage of update statements increases, the Recovery Manager needs to perform more code generation recovery updates. In turn, their performance overhead becomes an increasing cost for the log replay process and results in less throughput.

Callgrind results in Tables 4.8 and 4.9 show that an update operation is also spending most of its time on query execution (83.5%) and creating the Storage Interface (72.5%). On the other hand, the update operation (`StorageInterfaceTableUpdate`) accounts for only 20.46%. This

reflects that while updating a table is fast, the update replay is also burdened by execution initialization overhead. In fact, all the operations (insert/update/delete) have similar IR% for `StorageInterfaceInit`. This shows that this overhead is consistent and remains unaffected by the type of replay operations.

Similar to insert replays, overhead from value copies and execution context initialization accounts for 16%. Again, this overhead is unavoidable because the execution engine needs the execution context to run the update replay query, and the Recovery Manager needs the values to create an update physical plan.

We look into the implementation differences between baseline update replay and code generation replay to comprehend previous results. To perform an update replay in baseline recovery, the Recovery Manager performs the following operations: (1) initialize a new redo record for update by copying from the original redo record, (2) update the table with the new redo record, and (3) update indexes. In step (1), baseline recovery only needs to copy the redo record. While in code generation recovery, it needs to extract and copy values as we explained earlier for insert replays. Step (2) is equivalent to `StorageInterfaceTableUpdate`. On the other hand, code generation recovery still needs to construct/destruct the storage interface object, and this step takes up 79.25% IR%.

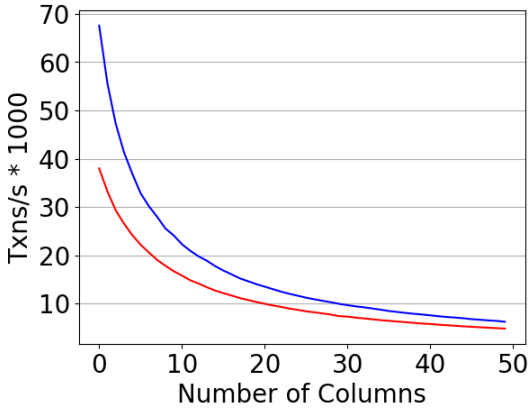
In summary, combined with results from Sections 4.1.1 and 4.1.2, we confirm that execution preparing/executing queries is the main bottleneck of code generation recovery. During replay execution, the execution engine is repeatedly creating and removing the storage interface; this results in reduced recovery throughput. In NoisePage, the execution initialization overhead comes from creating the storage interface object. To reduce this execution initialization overhead, however, we need to rewrite the storage layer of NoisePage. Therefore, this is out of scope for our discussion.

It is not surprising for baseline recovery to be faster than code generation recovery. One of the benefits of physical logging is that it provides fast recovery compared to logical logging [12]. The DBMS can update the database with simple table functions (index/delete/update). On the other hand, code generation recovery is similar to logical logging in that it requires the DBMS to re-execute the query (physical plan) during recovery. This means that unless code generation recovery has access to certain functions that are more efficient than those implemented in baseline recovery, it is impossible for code generation recovery throughput to catch up with baseline recovery throughput. We explore some of those functions in the next sections.

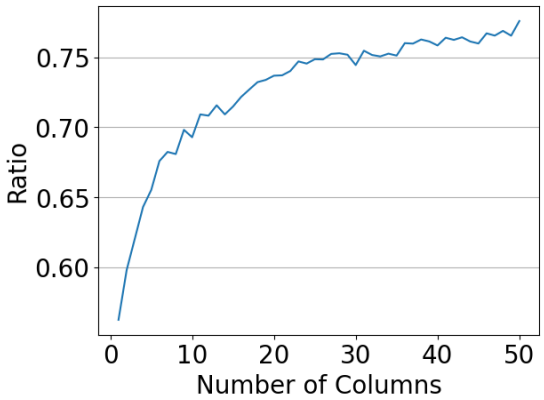
4.2 Index Reconstruction

Most in-memory DBMS do not log indexes to disk [5, 7, 29, 41]. Instead, these DBMSs reconstruct indexes during log replay. However, this introduces design problems for an execution engine that interacts 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. This custom index update function first locates the indexes for a log record's table. It then loops over the indexes and performs the following operations for each index: (1) rebuild a projection row for the index, (2)



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



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

Figure 4.1: Recovery with Indexes.

Functionality	IR%	Normalized IR%	Description
Index Update	96.23	96.3%	Update indexes affected by the insert replay.
Insert	3.11	3.11%	Execute an insert replay.
Other (e.g., tuple map access)	0.57	0.59%	

Table 4.10: Callgrind breakdown on Baseline Index Recovery. The associated table has 10 indexes. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

compute offset and copy the each value from the log record's table into the projected row, and (3) insert/delete the projected row from the index.

The code generation approach, however, uses the DBMS's built-in execution engine for index updates. While both the baseline index update and the execution index update implementations follow the same logic, the execution version receives additional the benefits exclusive to code generation. By compiling the index update function, the DBMS reduces function jumps and branch mispredictions within the index update loop. This improves the efficiency of step (2) for index update. Therefore, we believe that as the number of indexes increases, the performance gap between baseline and the code generation approach will diminish.

For this experiment, we 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, some of the overhead is still unavoidable (e.g., insert value copies, Storage Interface construction/destruction). We create an empty table with a single column with an **`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 in Figures 4.1a and 4.1b show that as the number of indexes increases, the gap be-

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%	

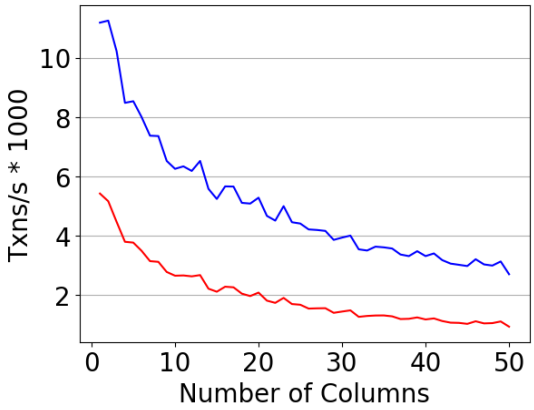
Table 4.11: Callgrind breakdown on a Query Execution step of Code Generation Index Recovery. The target table has 10 indexes. The query execution step accounts for 96% IR for a single replay step. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

tween the baseline and code generation implementation steadily decreases. Note that without any indexes, the gap between baseline and code generation is around $2\times$, which matches the results from Table 4.1. To verify whether the decreasing gap is due to difference in implementation, we use Callgrind to breakdown both baseline recovery and code generation recovery. The Callgrind results show that index updates account for 96.3% of total instructions in the baseline implementation (Table 4.10). On the other hand, for the code generation approach, index update accounts for $37.9\% + 23.5\% = 61.4\%$ of total instructions (Table 4.11). This percentage difference shows that code generation recovery uses a more efficient index update function. We also noticed that as the number of indexes increases, the overhead from the `StorageInterfaceInit` becomes less significant. In Table 4.11, it accounts for only 36.2 IR%, compared to the index insert step that takes up 37.9 IR%. This is because this initialization overhead does not scale with the number of indexes. As a result, it gives code generation recovery an advantage when the number of indexes is high; it is able to compensate for the execution initialization overhead by using a more efficient index update function.

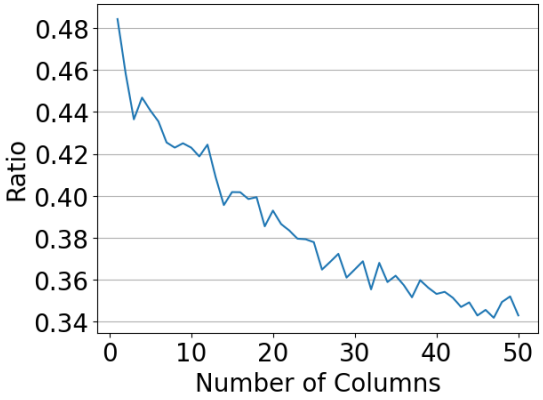
In conclusion, our analysis indicates that the code generation requires less overhead to prepare for an insert index operation. Hence, if the recovery execution can maintain a constant execution initialization overhead, code generation recovery can potentially outperform baseline recovery when the table has large number of indexes by compensating its initialization overhead with a more efficient index update implementation. We further verify this in the next experiment.

4.3 Scaling Number of Columns

Our test results with indexes have shown the potential strengths of code generation recovery. Besides indexes, we believe code generation recovery will also be more efficient compared to baseline recovery on a table with a higher number columns. The execution engine has access to more efficient function implementations compared to the Recovery Manager. If we scale the



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



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

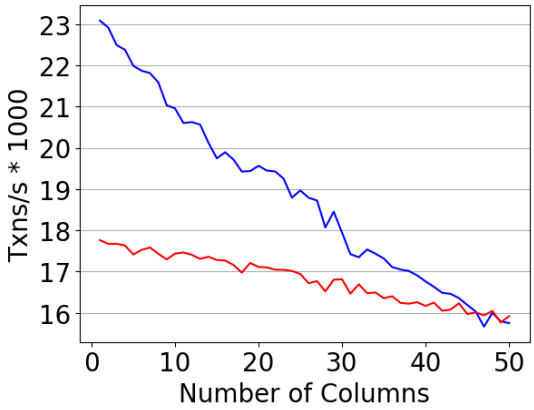
Figure 4.2: Recovery (100% Insert) with Scaling Number of Columns. As the number of columns increases, the throughput difference between code generation recovery and baseline recovery increases.

Functionality	IR%	Normalized IR%	Description
Query Execution	47.78	50.13%	Executes the update replay query.
Retrieving Record Values	45.84	48.09%	Copy values from a redo record into a physical plan.
Execution Context Initialization	1.37	1.44%	Initializes an execution context (Figure 3.13) for query execution.
Cache Access	0.31	0.33 %	Reading and writing values/queries into hash table caches.
Compile	0.01	0.01 %	Compiling the update replay query.

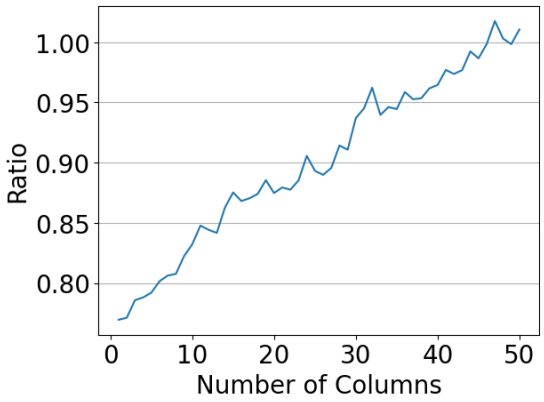
Table 4.12: Callgrind breakdown on an Insert Recovery step with 50 columns. IR (instructions read) refers to the number of instructions read in total by the selected function during a certain function execution.

table with higher number of columns, this can potentially reduce the performance gap between the two implementations. To verify our assumption, we performed two tests: (1) An insert recovery test with the same setup as in Section 4.1.1 that scales with the number of columns, and (2) an index recovery test with the same setup in Section 4.2, but we fix the number of indexes to be 10 and scale the table by the number of columns.

Results from the insert test (Figure 4.2) show that code generation recovery failed to shorten the throughput performance gap. Our Callgrind analysis (Table 4.12) shows that as the number of columns increase, the value copies overhead of an insert replay also increases. At 50 columns, value copies account for 48.09 IR%, where query execution only accounts for 50.13 IR%. This is because the value copies overhead scales with the number of columns. For a code generation insert/update replay, it needs to go over each value in the projected row, cast the value to the correct type, and copy that value into a new array to insert into a physical plan. Therefore, if there are more columns, then the value copies step needs to iterate over more values. On the other hand, baseline recovery does not contain value copies step. It only needs to copy the



(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: Index Reconstruction with Scaling Number of Columns. The associated table uses 10 indexes. As the number of columns increases, the throughput gap between code generation recovery and baseline recovery decreases.

entire redo record to generate a new log record. While the overhead of this copy still scales with the number of columns, baseline recovery does not need to iterate over every column of a log record. This difference in implementation leads to a wider gap between baseline recovery and code generation recovery as the number of columns increases.

The index recovery test showed that for index recovery, a higher number of columns have a positive effect in decreasing the gap between code generation recovery and baseline recovery (Figure 4.3). As the number of columns approaches 50, code generation index recovery and baseline index recovery reaches the same level of performance. As we have shown in Section 4.2, code generation uses a more efficient index update function, as it is a compiled version of the index update in baseline recovery. Therefore, while both the overhead of both implementations scales with the number of columns, code generation index update will incur less overhead in the long run because it is more efficient by reducing branch mispredictions and avoiding function jumps within loops. Combined with the results from Tables 4.10 and 4.11, we conclude that the gap shortens because the Recovery Manager has access to a more efficient implementation of index update.

Chapter 5

Related Work

In this section, we discuss DBMSs with recovery systems and code generation architectures that differ from our method assumptions. We then show whether and how our code generation approach applies to them.

5.1 Recovery

To begin with, our method does not extend to DBMSs that do not have logging architectures. We cannot apply our log record conversion logic to those systems. For instance, Facebook's Scuba [17] proposed a recovery system that uses shared memory to persist data between processes. On database shutdown, Scuba copies data directly from memory heap to shared memory. On restart, Scuba retrieves stored DBMS memory from shared memory. This recovery system does not depend on log records, so we cannot apply our approach to such systems.

Our code generation approach focuses on DBMSs that use physical logging for recovery. However, there are DBMSs that use other logging schemes, such as logical logging and physiological logging [19]. For instance, VoltDB [29] uses a variant of logical logging known as command logging. The DBMS records transactions instead of physical table changes in the log records. Nevertheless, those log records still correspond to insert, delete, or update operations on the DBMS table. Therefore, we can still apply our log record conversion process to VoltDB's recovery framework. However, since the log records already contain transactions, there is no need for the recovery system to prepare them for code generation. The DBMS can simply execute the transactions to proceed with recovery.

Amazon Redshift stores raw SQL statements [20] executed by users in its log records. In this case, there is also no need for the DBMS to use our code generation recovery approach, because the DBMS can directly parse and execute stored SQL statements with its execution engine.

MySQL uses physiological logging that stores database page number, operation code (e.g. insert/delete/update at a offset), and data tuples [8]. These log records contain sufficient information to support code generation recovery: the operation code allows the recovery system to

construct corresponding physical plans, and the data tuples provide values for these physical plans.

In differential logging [26], the recovery system stores log records that represents the difference between a before image (a copy of the table record before it is changed by a transaction) and an after image (a copy of the table record after it is changed by a transaction). The DBMS uses an XOR function to compute the difference between the before and after images and stores that difference into a log record. While differential logging is a form of physical logging, we cannot apply our code generation recovery approach to this logging scheme. Since each log record only contains differential information, it is impossible for the recovery system to interpret an individual differential log. Therefore, there is no way to convert a differential log into a physical plan.

Many in-memory DBMSs also implement parallel recovery to reduce recovery time [12]. Hekaton [7] uses a thread per core architecture to handle parallel insertion of the data from log records. SiloR [41] parallelizes recovery by using an epoch system and value logging that allows the recovery system to process log records in arbitrary order. During replay, the recovery system uses multiple log processors to replay log records from newest to oldest. Our code generation recovery approach might introduce extra overhead for parallelization This is because.the recovery system needs to access multiple hash tables (Section 3.4) that are necessary for executing compiled queries. This means that with multiple recovery threads, they need to read and modify those hash tables concurrently. All of the hash tables use keys based on database and table ids. Unless the recovery threads manage to operate on separate tables, the recovery system needs to install extra latches on the hash tables. As as result, this incurs performance bottlenecks for parallel recovery.

DBMSs use checkpoints for their recovery systems to reduce database recovery time [7, 38, 41]. Since NoisePage does not support checkpoint recovery, we did not mention how we would incorporate checkpoint for code generation recovery. With checkpoint recovery, the database restores itself to a previous checkpoint and replays log records afterwards [32]. To incorporate code generation recovery, a DBMS needs to wait after it loads the checkpoint. The DBMS can then start converting log records into physical plans for execution as described in our approach. For in-memory DBMSs, loading checkpoints requires the DBMS to copy large data blocks from disk into memory. The DBMS then scans the tables to rebuild indexes. In this case, the recovery system still needs to use/implement an index update function. While code generation recovery cannot reduce data copy overhead, it allows the DBMS to use the built-in index update in the execution engine for this index reconstruction step.

5.2 Code Generation

NoisePage code generation uses an embedded compiler (LLVM) running in the same process [31]. However, there are systems that depend on external compilers for code generation (e.g., Ama-

zon Redshift, HIQUE, LegoBase). In those systems, the DBMS emits source code to an external compiler that generates machine code [23, 25]. Our code generation recovery approach is agnostic to the compilation layer of the DBMS. It only requires changes to physical plans and execution functions. HIQUE converts physical plans into query- and hardware-specific C source code and compiles with gcc compiler into a shared library file [25]. The query executor then dynamically loads the shared library file to evaluate the query. To implement code generation recovery for HIQUE, we convert log records into HIQUE's physical plans. Since the shared library file is linked with the DBMS, we can modify functions within HIQUE's execution engine to complete the implementation for code generation recovery.

Amazon Redshift [20] rewrites the query plan to generate compiled C++ code on the leader node of a database cluster. The leader node then sends the compiled binaries and query parameters to compute nodes. The compute nodes execute the binary with the parameters and send results back to the leader node for final aggregations. This means that the leader node will be responsible for updating the final physical contents of the database. To use code generation recovery, we first provide the leader node with our converted replay physical plan. We provide additional information (e.g., tuple slot) as parameters. The leader then performs the execution as usual to finish log replay.

In our implementation, we also showed that because NoisePage converts physical plans into operator pipelines, we need to modify its operator pipelines to ensure correctness. NoisePage implements operator pipelines to delay tuple materialization and allow a data tuple to stay in the CPU registers as long as possible [33]. However, our code generation recovery approach does not depend on the concept of operator pipelines that is specific to NoisePage. We must modify the operator pipelines in NoisePage because they are interpretations of physical plans. We now describe how we can apply code generation to DBMSs that transforms physical plans into other representations [10, 15, 20, 23, 25, 34]. Hekaton [15] uses an imperative syntax tree to generate C code. It then compiles the C code with Visual C/C++ compiler and linker to generate a DLL file. An OS loader then links the DLL file at runtime. The imperative syntax tree is generated from a mixed abstract tree (MAT) that is able to represent query plans [15]. The functionality of MATs is close to that of the physical plan nodes in NoisePage. Therefore, to apply our code generation recovery, the DBMS can start by converting log records into MATs. Because the imperative syntax tree is an interpretation of the MATs, we also need to modify the conversion process from MAT to impertaive syntax trees. Finally, we modify execution functions inside the DBMS's execution engine to pass in additional information (e.g. tuple slots) during execution.

SingleStore [10] converts physical plans into MemSQL Plan Language (MPL) that is designed specifically for SingleStore. The DBMS compiles MPL into machine code with LLVM. Since the DBMS generates MPL from physical plans, we need to modify the conversion process from physical plans to MPL. Another approach is to convert log records direclty into MPL. This is because MPL relational operations (e.g. index seek, scalar operations) and is sufficient for representing a log record. The DBMS then only needs to change its execution engine to support code generation recovery.

Oracle Timesten [34] does not transform its physical plans into other representations, but passes them directly to its code generator for compilation. The only changes we need to make then is to convert the log records into physical plans and modify the DBMS's execution engine.

Chapter 6

Conclusion and Future Work

We presented a code generation recovery approach for in-memory DBMSs. Our method integrates the code generation execution engine with the recovery system by converting log records into physical plans and compiling them into 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 throughput experiments. The results show that while code generation recovery has worse recovery throughput compared to NoisePage's baseline recovery, it offers less engineering overhead and is more efficient in processing index updates.

We confirmed that the execution initialization for compiled queries is the main bottleneck for code generation recovery in NoisePage. A possible solution to this is to cache reusable objects that are repeatedly initialized during query execution. We will explore on more solutions to this problem and continue to improve our code generation recovery based on our current implementation in NoisePage. We also believe it will be valuable to evaluate our code generation recovery against a recovery system that implements a logical logging scheme (e.g., VoltDB). We explained in the Related Works section that it is excessive for DBMSs that use logical logging to adopt code generation recovery. On the other hand, physical logging allows for faster recovery compared to logical logging, but it incurs extra engineering effort for query compilation DBMSs. Code generation recovery serves as a middle-ground between logical and physical logging: it removes the extra engineering effort for physical logging, but also slows down the recovery process. Hence, a comparison between logical logging recovery and code generation recovery will provide us with important insights on how to further improve code generation recovery.

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, 4.2

[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] 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, 4.2, 5.1

[8] MySQL Documentation. Web, 2021. 2.1, 5.1

[9] PostgreSQL Documentation. Web, 2021. 2.1

[10] SingleStore Documentation. Code generation. https://docs.singlestore.com/db/v7.3/en/query-data/run-queries/query-concepts/code-generation.html, 2021. 5.2

[11] Valgrind Documentation. Callgrind. https://valgrind.org/docs/manual/cl-manual.html, June 2021. 4

[12] 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, 4.1.3, 5.1

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

[14] 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

[15] Craig S. Freedman, Erik Ismert, and P. Larson. Compilation in the microsoft sql server hekaton engine. *IEEE Data Eng. Bull.*, 37:22–30, 2014. 5.2

[16] 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

[17] 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, 5.1

[18] 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

[19] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902. 5.1

[20] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1917–1923, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2742795. URL https://doi.org/10.1145/2723372.2742795. 5.1, 5.2

[21] 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

[22] 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

[23] 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, 5.2

[24] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 197–208, 2018. doi: 10.1109/ICDE.2018.00027. 3.2.1

[25] 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, 5.2

[26] Juchang Lee, Kihong Kim, and S.K. Cha. Differential logging: a commutative and associative logging scheme for highly parallel main memory database. In *Proceedings 17th International Conference on Data Engineering*, pages 173–182, 2001. doi: 10.1109/ICDE.2001.914826. 5.1

[27] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. *Proc. VLDB Endow.*, 14(4):534–546, 2020. URL https://db.cs.cmu.edu/papers/2020/p534-li.pdf. 3.1.1

[28] Google LLC. Google Benchmark. https://github.com/google/benchmark, June 2021. 4

[29] 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, 4.2, 5.1

[30] 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, 3.2.1

[31] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, 14(2):101–113, October 2020. ISSN 2150-8097. doi: 10.14778/3425879.3425882. URL https://doi.org/10.14778/3425879.3425882. 5.2

[32] 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, 3.1.3, 5.1

[33] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011. 1, 2.2, 3.2.1, 5.2

[34] Oracle. Query optimization. https://docs.oracle.com/cd/E11882_01/timesten.112/e21631/query.htm#TTCIN193, 2021. 5.2

[35] 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

[36] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017. URL https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf. 3.1.3

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

[38] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. Low-overhead asynchronous checkpointing in main-memory database systems. SIGMOD '16, page 1539–1551, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2915966. URL https://doi.org/10.1145/2882903.2915966. 5.1

[39] 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

[40] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10: 781–792, March 2017. URL https://db.cs.cmu.edu/papers/2017/p781-wu.pdf. 3.1.1

[41] 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, 3.1.1, 4.2, 5.1