Master Project Report

Obsidian: a fully decentralized indexer solution for the Sui network

Loris Tran and Alexandre Mourot
MSc in Computer Science
École Polytechnique Fédérale de Lausanne

Rachid Guerraoui
Thesis Advisor

Gauthier Voron
Thesis Supervisor

Distributed Computing Laboratory
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne

May 2025

Abstract

This master project addresses a significant challenge in the Sui blockchain ecosystem: the time-consuming and resource-intensive process of data indexing. Our research reveals that development teams spend approximately 30% of their project time implementing indexing solutions to access and analyze on-chain data and that accessibility to the indexing framework.

We introduce two complementary contributions to address this bottleneck:
(1) a modular SDK that allows developers to specify precise actions on specific transaction fields, and (2) a generic indexer that automatically processes all transactions related to a specific package without requiring deep knowledge of the Sui framework.

Both solutions prioritize data self-custody, allowing teams to run indexers locally or in their own infrastructure. Our implementations significantly reduce the technical barrier to entry for data indexing on Sui, enabling development teams to focus on their core application logic while maintaining full control over their data. Comparative analysis with existing solutions demonstrates the efficiency and accessibility advantages of our approach, particularly for teams without specialized blockchain knowledge or dedicated indexing resources.

1 Introduction

In the rapidly evolving landscape of blockchain technology, the ability to efficiently access and analyze on-chain data has become a critical requirement for decentralized applications (dApps). As blockchain ecosystems mature, the volume of transaction data grows exponentially, creating significant challenges for developers who need to access this data in real-time to power their applications.

The Sui blockchain, developed by Mysten Labs, represents a new generation of high-performance blockchains designed for wide-scale adoption. With its object-centric data model and high throughput capabilities, Sui enables developers to build sophisticated dApps with enhanced performance and scalability. However, this advancement comes with its own set of challenges, particularly in the realm of data indexing and retrieval.

1.1 The Critical Role of Indexers in Blockchain Ecosystems

Blockchain indexers play a pivotal role in the ecosystem by transforming raw on-chain data into structured, queryable formats that applications can easily consume. Without efficient indexing solutions, dApps would need to scan the entire blockchain history to retrieve relevant information each time they need it or make inefficient RPC calls to gather the specific data they need, a process that is both resource-intensive and time-prohibitive.

On the Sui blockchain, indexers are especially important due to several factors.

Modern dApps require near-instantaneous reflection of on-chain state changes in their user interfaces, making real-time frontend updates crucial.

Additionally, teams depend on comprehensive analytics and insights regarding user interactions, transaction patterns, and contract performance to optimize their applications, communicate data and adapt their strategy.

Access to historical data is another key requirement, as many applications need transaction histories that full nodes may not readily provide.

Furthermore, each application may have unique requirements for filtering and processing on-chain modification, necessitating customizable indexing solutions tailored to specific use cases.

1.2 Current Challenges in Data Indexing on Sui

Indexing data in blockchain applications presents several challenges.
Building a custom indexer demands advanced Rust skills and deep familiarity with the Sui framework, posing a steep learning curve. Traditional indexing methods often require significant development resources, pulling focus from core product work. Many third-party solutions lack self-custody and full data ownership, introducing dependency risks. Finally, the cost of building and maintaining custom infrastructure can be a major barrier, especially for early-stage teams.

Taking insight from the Sui DevRel team, founders and incubated project on Sui, revealed that implementing proper data indexing solutions consumes approximately 30% of the total development time required to bring a product to market on the Sui blockchain. This represents a significant bottleneck in the development pipeline and highlights the urgent need for more efficient and accessible indexing solutions.

1.3 Towards a More Accessible Indexing Solution

This master project addresses these challenges by introducing two complementary contributions to the Sui ecosystem:

1. A modular SDK approach that allows developers to specify precise actions on specific fields of checkpoint transactions, simplifying the framework and reducing the learning curve.

2. A generic indexing solution that automatically fetches all transaction data related to a specified

package, enabling teams to run analytical operations without deep knowledge of the Sui framework.

Both solutions are designed with self-custody in mind, allowing teams to maintain full ownership of their data by running the indexer locally or in their own infrastructure. By providing these tools, we aim to significantly reduce the development overhead associated with data indexing, allowing teams to focus on their core application logic while still benefiting from comprehensive on-chain data access.

The following sections will detail the technical implementation of these solutions, explore their benefits compared to existing approaches.

2 Background and Current State of Indexing on Sui

To understand the challenges and opportunities in Sui blockchain data indexing, it is essential to first establish a clear understanding of the Sui architecture, data model, and current indexing approaches.

2.1 Sui Blockchain Architecture

Sui is a high-performance Layer 1 blockchain designed with an object-centric data model. Its programming language, Move, is derived from Rust and enforces this object-centric paradigm.

Sui differentiates itself from account-based blockchains like Ethereum. In Sui, the fundamental unit of storage is an object rather than an account. Each object has a unique identifier (ID) that persists throughout its lifetime. Objects either belong to a specific address or are shared objects accessible by multiple participants.

Understanding how data is stored and modified is key to understanding the need for indexing. All on-chain state is represented as objects that can be created, modified, or deleted through transactions.
These transactions are grouped into checkpoints, which are created every 2–3 seconds. Checkpoints serve as units of finality for transactions. Each checkpoint includes signatures from a supermajority (at least 2/3) of validators, attesting to the validity of the included transactions.
Sui also leverages parallelism to execute transactions that do not interact with shared objects, enabling significantly higher throughput.

This architecture enables high performance but also introduces unique challenges for data indexing compared to traditional account-based blockchain architectures. Indeed to track objects belonging to a certain package we may need to retrace it's history throughout all the transactions that interacted with it whereas in an account based blockchain we would have just scanned the smart contract state that defines this object

2.2 Data Flow and Transaction Lifecycle in Sui

2.2.1 Transaction Structure

A typical Sui transaction proceeds through several stages. First, a user creates a transaction by specifying the objects to be read, modified, or created.

The transaction is then signed by the sender and, if necessary, by additional stakeholders. After signing, the transaction is submitted to the validators, who execute it and produce the corresponding effects.

Finally, the transaction is finalized when included within a checkpoint that gathers the signatures of 2/3 of the network. This is the unit of finalization.

To avoid any confusion, it's worth mentioning that epochs of 24h are used for system update and not as a unit of finality. i.e update of the staking amount and validators.

Notably, Sui enables fast-tracking of transactions that only involve owned objects. This design allows such transactions to bypass consensus, leading to significantly higher throughput and reduced latency.

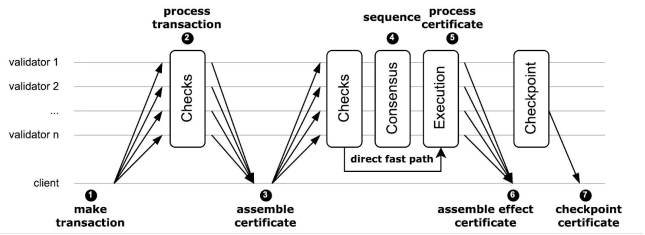

Figure 1: Sui transaction lifecycle

Each transaction in Sui produces a detailed dataset that serves as the foundation for indexing and extracting wanted data. Indexing this data enables faster and more efficient access to application-relevant information.

The Rust structure defining transactions inside a checkpoint is the following:

```
pub struct CheckpointTransaction {
    /// The input Transaction
    pub transaction: Transaction,
    /// The effects produced by executing
        this transaction
    pub effects: TransactionEffects,
    /// The events, if any, emitted by
        this transactions during execution
    pub events: Option<TransactionEvents>,
    /// The state of all inputs to this
        transaction as they were prior to
        execution.
```

```
    pub input_objects: Vec<Object>,
    /// The state of all output objects
        created or mutated or unwrapped by
        this transaction.
    pub output_objects: Vec<Object>,
}
```

Listing 1: CheckpointTransaction struct in Rust

The transaction's fields serves specific purpose and are a choice of design that is both efficient and practical to retrieve in formations depending on use cases. The events field is crucial for frontend logic like it's often the case with blockchain, there is also object oriented specific fields like the input and output object that are crurcial to keep object's history.

Each transaction is uniquely identified by its transaction digest, which is derived from the hash of its content.

2.2.2 Checkpoint Structure

Checkpoints are produced every 2–3 seconds and group all transactions executed during that interval. They are the core unit of data organization in Sui and play a central role in how data is finalized and accessed.

Each checkpoint includes a sequence number to identify its position in the ledger and a digest representing the hash of its content. It also contains the list of transaction digests it finalizes and a reference to the previous checkpoint digest, ensuring continuity and consistency in the chain. Additional metadata is included, such as validator signatures and consensus commitments.

From an indexing perspective, checkpoints define clear batch boundaries. Indexers typically process data one checkpoint at a time, which simplifies ingestion and ensures ordering. This structure is essential for maintaining reliable and scalable indexing pipelines aligned with Sui's execution and finality model.

2.3 Data accessibility on Sui

The figure 2 illustrates the current architecture for accessing and processing data from the Sui network.

At the base layer, Sui Full Nodes expose gRPC endpoints, which serve data directly from the Sui base API. These endpoints can be queried by data consumer applications—ranging from individual developers to full-scale apps—allowing them to retrieve blockchain data.

For more efficient and flexible access to specific datasets, developers can implement a custom indexing solution. This involves processing the raw data, transaction checkpoints explained previously, through an Indexer, storing the results in a database and providing RPC endpoint access to data consumer.

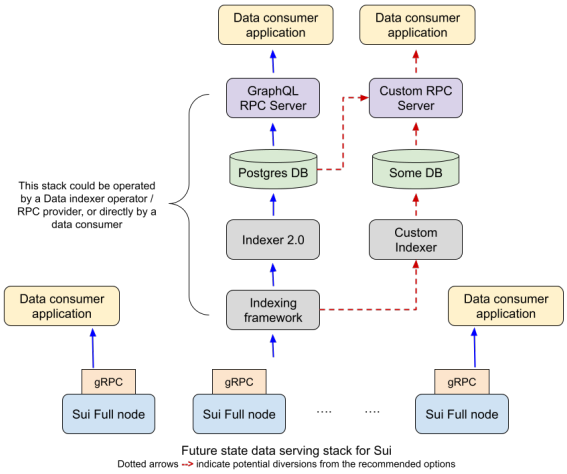

Figure 2: Sui Data accessibility Schema

This layered architecture incorporating an indexer enables more targeted and performant data queries, which are particularly necessary for analytical tools or applications requiring specialized views of blockchain activity.

2.4 Current Indexing Methods and Schemas

2.4.1 Native Sui Indexing

The Sui framework provides basic data access through its Full Node API, which supports RPC calls for interacting with the blockchain state.

However, this native approach has several limitations for dApps. It lacks support for complex or customized queries, and performance is not optimized for analytics or high-frequency data access.

Additionally, historical data older than 2 to 5 epochs, depending on the configuration of each Full Node, is pruned to prevent the database from growing to an unmanageable size. When such pruned data is queried, the RPC must rely on archival nodes, which consequently increases latency

Most notably, the system does not support custom indexing, making it difficult for developers to isolate and store only the data relevant to their specific applications.

2.4.2 Current Third-Party Indexing Solutions

To address the shortcomings of native indexing, several third-party solutions have been developed:

- **Sentio:** Offers an SDK for building custom indexers with a monthly subscription model. However, it requires using Sentio's cloud infrastructure and database hosting, limiting data self-custody.

- **Custom Rust Indexers:** Many teams build their own indexers using the Sui Rust SDK for full control and flexibility.

2.5 Limitations and Challenges of Current Approaches

Through discussions with the Sui Developer Relations team and projects in the Sui Accelerator, we identified key limitations in current indexing approaches that collectively create a significant barrier to entry, especially for early-stage projects.

Implementing custom indexers is technically demanding. Developers must have a deep understanding of the Sui framework, Rust, and database architecture. These systems require ongoing maintenance, diverting time and resources away from core product development. Moreover, existing third-party solutions often compromise data sovereignty, forcing teams to relinquish control over their data and introducing risks such as vendor lock-in. Existing solutions are under a SaaS form and need a subscription payment.

Beyond these technical and operational hurdles, our research highlights a broader developer experience gap. There is a notable absence of accessible, self-custodial indexing solutions within the Sui ecosystem. As a result, teams must either invest heavily in building and maintaining their own infrastructure or rely on external providers. On average, nearly 30% of development time is spent on indexing-related tasks. Limited resources can also lead teams to deprioritize data accessibility, which in turn hampers analytics, monitoring, and dashboard development that are necessary to a good project development.

This experience gap is not only a technical constraint but also a strategic opportunity to strengthen the Sui developer ecosystem and foster support for new applications. In the following sections, we outline our approach to addressing these challenges through two complementary contributions.

3 Contribution 1: Modular Indexer SDK Approach

Our first contribution addresses the complexity of data indexing on Sui through a modular SDK approach. This implementation, found in the `sui-indexer-modular` repository, provides a flexible framework that allows developers to specify both the database type and precise actions on specific fields of checkpoint transactions.

3.1 Design Philosophy

The design philosophy of our modular indexer SDK is built around three key principles that prioritize control, flexibility, and ease of use.
First, selective processing allows developers to choose exactly which transaction fields, from a certain package, they want to process and store, avoiding unnecessary overhead.
Second, customizable actions enable the use of custom logic for handling transaction data fields, allowing developers to extract only the relevant information needed for their use case.
Third, self-custody ensures that all data is stored in the developer's own database, in their preferred format, giving them full control and ownership of the indexed data.

This design reduces the complexity typically involved in building custom indexers. The SDK is easy to adopt and removes the need to manually research and integrate multiple repositories, while still offering full flexibility to meet project-specific needs. It is built on top of the sui-indexer-alt-framework developed by Mysten Labs.

3.2 Technical Implementation

3.2.1 Architecture Overview

The modular indexer SDK is implemented as a Rust library that leverages the Sui Alt Framework for checkpoint processing and wrap the classic indexer provided by Mysten. The architecture consists of several key components:

- **SuiIndexer:** The main entry point that coordinates the indexing process.

- **IndexField:** An enum defining the various data fields from a transaction that can be processed and indexed : (transactions, effects, events, input objects, output objects).

- **IndexCallback:** A type for callback functions that process specific fields of checkpoint data and extract the indexed data.

- **IndexerPipeline:** The processing pipeline that filters transactions and applies callbacks.

- **Database Schema:** A structured PostgreSQL schema that stores the indexed data.

The general structure of the indexer implements the following pattern:

```
pub struct ModularIndexer<T> {
    package_filter: Option<SuiAddress>,
    field_filters: Vec<IndexField>,
    field_callbacks: HashMap<IndexField,
    FieldCallback<T>>,

    indexer: Option<IndexerCluster>,
}
```

3.2.2 Core Components Analysis

The core implementation revolves around several key aspects that provide flexibility and control over the indexing process.

The SDK provides a clean interface for selecting which fields to index from checkpoint's transactions, allowing developers to specify exactly what data they need to process:

```
indexer.set_filter_fields(vec![
    IndexField::Transaction,
    IndexField::Effects,
    IndexField::Events
]);
```

For each selected field, developers can register a custom callback function that processes the data according to project-specific needs. This enables tailored data transformation and filtering:

```
indexer.set_filter_callback_for_field(
    IndexField::Transaction,
    |checkpoint_data| {
        // Custom processing logic
        Ok(processed_data)
    }
);
```

The SDK allows focusing on transactions that interact with specific packages, reducing processing overhead and improving efficiency:

```
indexer.set_filter_package(package_address);
```

Finally, the SDK provides flexibility in database configuration by allowing developers to specify a custom database URL for storing the indexed data:

```
indexer.
start(args.database_url,args.cluster_args);
```

3.3 Database Schema Design Using Supabase

The modular indexer uses a PostgreSQL database integrated with Supabase.

Users define their database tables and types in a .sql file, and running a Diesel migration will create the corresponding database schema.

We provide a fully integrated solution with Supabase, an open-source framework for database management that offers automatic API endpoint generation. These endpoints can be used across the entire architecture. By running our Docker setup, users can automatically deploy a Supabase instance, which can be queried through API calls and managed using the graphical interface provided by Supabase.

This setup simplifies database management, streamlines API integration, and enhances the developer experience.

3.4 Local vs. Remote Mode

The generic indexer supports two distinct operational modes: Remote Mode and Local Mode.

In Remote Mode, the indexer connects to a Sui Bucket maintained by Mysten Lab on AWS to fetch checkpoint data, making it suitable for indexing live data. This is especially useful for projects that do not want to run their own Sui Full Node. However it can increase latency as the indexer relies on Rpc calls to this bucket in order to fetch each checkpoint.

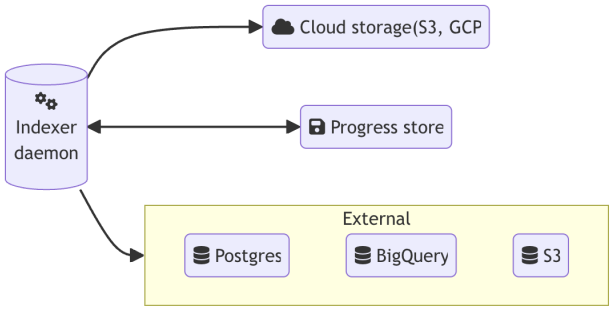

Figure 3: Remote ingestion Indexer Schema

Local Mode (see fig 3), for team using our docker solution to run a Sui Full Node, they can process checkpoint files stored locally. This ideal for historical data analysis. Indeed it allows for way faster ingestion compared to remote ingestion mode.

This flexibility allows teams to choose the most appropriate mode for their specific requirements.

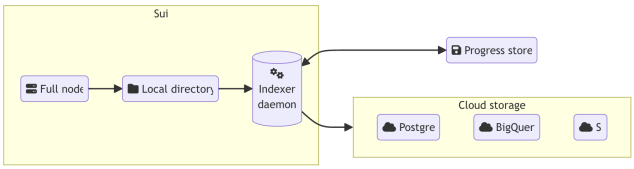

Figure 4: Local Ingestion Indexer Schema

3.5 Implementation Walkthrough

The implementation offers a developer-friendly workflow for setting up and running the Sui indexer efficiently.

The developers configure a package filter to specify which smart contract package should be tracked. After selecting the relevant package, they choose which transaction fields to index.

Custom callbacks are then registered to define how each of these selected fields should be handled. These callbacks encapsulate the logic needed to extract meaningful data from the transactions.

To enable persistent storage, the database URL must be provided, indicating where the processed data will be saved.

Finally, the indexer can be started, beginning from a specified checkpoint. From this point onward, it continuously processes new live transaction data according to the defined configuration and logic.

Before starting the indexing process, developers must explicitly define the data types to be extracted from each selected field, such as specific event structures, transaction details, or object types.

Additionally, they need to establish the structure of their target database schema, ensuring that the extracted data can be stored and queried efficiently according to their application's requirements.

3.6 Advantages over Traditional Approaches

The modular SDK approach offers several key advantages over traditional custom indexer implementations.

First, it significantly reduces development time by providing a structured framework with well-defined extension points. This allows teams to build and deploy custom indexers more efficiently, without having to reinvent foundational components.

Second, it lowers the technical barrier to entry. Developers, particularly those working on early-stage projects, can focus on their specific data processing needs without requiring deep knowledge of the Sui indexer framework's inner workings. This streamlining accelerates prototyping and reduces onboarding complexity.

Finally, the SDK promotes data self-custody. Unlike third-party indexing services, it allows teams to retain full control over their data. They can store only the information that is relevant to their application, while preserving the ability to reconstruct the entire transaction history if necessary. This ensures both privacy and flexibility in data management.

Compared to Sui's native indexing approach, which often relies on repeated API calls, the modular SDK offers a significantly faster and more reliable solution. By directly processing on-chain data through a custom pipeline, it avoids the latency and rate limits associated with external queries, enabling more consistent and scalable indexing.

3.7 Limitations and Future Work

While the modular SDK significantly enhances the developer experience, it still has a few limitations.

Some knowledge of Rust is required to implement custom callbacks, and the process of creating types involves some boilerplate code.

Future work could focus on implementing automatic type derivation directly from Move smart contracts, allowing developers to seamlessly index defined structures without manually writing Rust types. Conversely, the system could also support deriving Rust types from predefined database schemas, streamlining the workflow for developers who start from a database-first design.

A similar approach has been taken by Buildy, which provides a tool for automatically generating TypeScript types for events directly from smart contracts. Adopting such automation would further reduce manual overhead and make the SDK more accessible to developers with varying levels of Rust expertise.

4 Contribution 2: Generic Package-Based Indexer

Our second contribution offers an alternative approach to data indexing on Sui through a so-called generic, package-focused indexer. This implementation, found in the `sui_indexer_checkpointTx` repository, provides a streamlined solution that automatically processes all transactions related to a specific package without requiring deep technical knowledge of the Sui framework. Unlike the modular SDK (sui-indexer-modular), this solution does not allow the developer to choose the database type or customize the data extraction directly during the checkpoint processing.

However, we provided a solution through Appsmith that allows anyone to create his own data dashboard Gui

out of the database. This can also be further customized using JavaScript, an easy and well-known programming language.

4.1 Design Philosophy

The generic indexer is built on a fundamentally different philosophy compared to the modular SDK. It emphasizes simplicity by minimizing configuration and coding requirements, making indexing accessible to all teams with minimal effort.

Another key aspect is its package-focused approach. The indexer automatically captures all data related to a specific smart contract package, removing the need for custom selection or filtering.

Deployment is also streamlined through containerization. The indexer includes ready-to-use Docker configurations, ensuring it can be deployed easily, just like other parts of our project.

Moreover, it supports both local and remote modes. This means teams can either process local checkpoints or fetch them remotely, following a similar pattern to the modular SDK see Figure 4 .

This design is ideal for teams that require quick access to raw on-chain data without having to deeply understand the Sui framework or invest time in building custom indexing tools. They can later process this data using any programming language that fits their stack or integrate it directly into data visualization platforms such as Appsmith or Dune.

4.2 Technical Implementation

4.2.1 Architecture Overview

The generic indexer is implemented as a lightweight Rust application with full Docker support. Its architecture mirrors the key components of our modular SDK, but with a simplified and opinionated configuration.

The main difference lies in the use of fixed parameters and predefined types for extracted data, making the generic indexer a specialized implementation of the modular SDK.

The target smart contract package is specified in advance, and all fields from each transaction are indexed to ensure complete access to raw on-chain data. Both the data types and database schema are fixed and preconfigured.

4.3 Deployment and Usage

One of the key advantages of the generic indexer is its streamlined deployment process. It offers a straightforward setup using a '.env' file, allowing the entire configuration to be defined through environment variables. For example:

```
PACKAGE_ADDRESS=0x123...def
REMOTE_STORE_URL=https://fullnode.mainnet.sui.io
START_CHECKPOINT=0
LOCAL_MODE=false
```

To deploy, users simply run a single Docker Compose command, which launches both the indexer and its associated PostgreSQL database. Once running, the indexed blockchain data becomes immediately accessible via standard SQL queries.
This Docker-based strategy removes the need for complex setup procedures and ensures consistent behavior across different environments, making it ideal for teams looking for a fast and reliable indexing solution.

4.4 Data Model and Storage

The generic indexer uses a PostgreSQL database integrated with Supabase. Like the modular SDK, it also filters data provenance from a specific package and extracts the following fields into the database:

- **transactions:** Stores transaction metadata and serialized transaction data.
- **transaction_effects:** Records the effects of transactions, including created, modified, and deleted objects.
- **transaction_events:** Captures custom events emitted during transaction execution.
- **input_objects:** Tracks objects used as inputs to transactions.
- **output_objects:** Records objects created or modified by transactions.

Each table stores the fields in JSON format, enabling flexible data storage and later querying. This allows for efficient queries while maintaining full data fidelity. The data can be queried and modified by users using Supabase API automatically-generated endpoints, making it easy to integrate with data dashboard GUIs such as Appsmith or Dune.

This approach sacrifices some of the structural rigidity provided by the modular SDK in favor of greater simplicity.

4.5 Advantages for Non-Technical Teams

The generic indexer provides significant benefits for teams with limited technical resources by removing the need for complex development or infrastructure

setup. One of its key strengths is the zero-code configuration: users only need to modify basic environment variables to begin extracting raw data from the blockchain, without writing a single line of code.

Deployment is also made simple through a turnkey Docker-based setup. This approach eliminates intricate installation procedures and allows teams to get up and running quickly with minimal technical overhead. Once deployed, the indexer grants teams full data sovereignty, enabling them to retain complete control over their data without relying on external service providers.

The containerized design ensures minimal ongoing maintenance, making it easy to keep the system running with little effort. At the same time, the architecture remains extensible, more advanced users still have the option to customize or expand the indexing process to meet specific needs.

These advantages make the generic indexer an ideal solution for teams that want to focus on their core business logic and product development, rather than dealing with the complexities of infrastructure and data engineering.

4.6 Limitations and Future Work

While the generic indexer excels in simplicity and ease of use, it does come with certain limitations. Compared to the modular SDK, it offers less flexibility in how checkpoint data is processed.
This design choice results in a trade-off between simplicity and efficiency: in some cases, the indexer must process the data twice rather than once, which can lead to increased overhead.
Additionally, because it is designed for general-purpose indexing, the system may process more data than is strictly necessary for specific use cases, potentially impacting resource efficiency. Storage can become a problem as well as computation overhead as a second processing on top would require to go through the transaction data once again.

Future development will focus on addressing these limitations while maintaining the core simplicity of the tool. Planned improvements include adding configurable data transformation options to allow for more tailored indexing behavior and implementing schema templates for common use cases, such as tracking the number of transactions.
Integration with popular visualization tools and platforms, such as Dune, will also be explored to enhance accessibility and usability for non-technical users.

5 Analysis and Comparison

This section presents a comparative analysis of our two indexing solutions, the Modular SDK and the Generic Indexer, against a no indexing approach only retrieving data from Sui RPC call. We will refer to the latter under Sui native solution. We focus on indexing speed, computational complexity, and feature set. Considering the storage capabilities, Supabase is running locally and therefore its capabilities scale with the hardware.

5.1 Indexing Speedup

Experimental results demonstrate that both the Modular SDK and Generic Indexer significantly accelerate the indexing process compared to the native Sui approach which is not suitable for the use we want with the data. Table 1 summarizes the observed speedup factors.

To evaluate the real-world performance of data retrieval, we conducted benchmarks comparing RPC-based checkpoint queries against database queries from our indexed data using the Generic indexer and the modular SDK.

Table 1 presents the results from 100 experiments, querying transaction data from a checkpoint using both our indexing solution or a simple RPC call. We assumed that the checkpoints where the data to index is located is known ahead and that we query the corresponding checkpoint from the RPC:

Table 1: Query Performance Comparison on 100 checkpoints queried

Metric	RPC	Database
Average Latency	36.21 ms	687.28 µs
Latency Variability	1.22	2.72
Speedup Factor	1x	52.69x

The database query approach demonstrates a significant advantage in both latency and predictability. While RPC calls take an average of 36.21 ms to retrieve checkpoint data, database queries complete in just 687.28 µs, representing a 52.69x improvement in query latency. This speedup makes the database solution particularly suitable for applications requiring to perform high-frequency queries on large amounts of checkpoints.

The latency variability metric, which measures the spread of response times relative to the average, shows that database queries maintain more consistent performance. With a variability factor of 2.72 compared to RPC's 1.22, the database approach provides more predictable latency. This can be explained as querying the data from the database doesn't depend on any other actor, whereas for the RPC calls we need to cope with network traffic load.

5.2 Computational Complexity

The computational complexity of data retrieval and indexing differs significantly between the native Sui approach and our proposed solutions. With native Sui Solution (assuming no caching), every query for application-specific data requires scanning the entire transaction history on all the checkpoints making repeated RPC calls. In this scenario we don't consider the size of all the checkpoints but it's important to note that it's not possible in practice.

For N checkpoints and Q queries, the total work is $O(NQ)$, as each query may need to inspect all N checkpoints to process transactions touching the package. In contrast, both the Modular SDK and Generic Indexer preprocess and store relevant data in a structured database. The initial indexing step is $O(N)$ (each checkpoint is processed once), but subsequent queries are direct lookups or indexed searches, reducing query complexity to $O(Q)$. This separation of concerns (one-time indexing, fast queries) is a key advantage for analytics.

Table 2: Computational Complexity Overview

Solution	Indexing	Query
Native Sui	–	$O(NQ)$
Modular SDK	$O(N)$	$O(Q)$
Generic Indexer	$O(N)$	$O(Q)$

5.3 Feature Comparison

Table 3: Feature Comparison of Indexing Approaches

Feature	Modular SDK	Generic Indexer
Custom Field Selection	✓	✗
Custom Processing Logic	✓	✗
Data Self-Custody	✓	✓
No-Code Setup	✗	✓
No Rust Required	✗	✓
Docker Deployment	✓	✓
Supabase Integration	✓	✓
Real-Time Indexing	✓	✓

Table 3 highlights the practical differences between the approaches and what concrete improvement it allows as an indexing tool.

Only the Modular SDK allows developers to precisely select which transaction fields to index and to define custom processing logic via callbacks, which is essential for advanced analytics and tailored data extraction. Both the Modular SDK and Generic Indexer store data in user-controlled databases, ensuring privacy and sovereignty, while the native Sui approach relies on external nodes and does not offer this guarantee.

The Generic Indexer is designed for less to non-technical users, requiring only environment variable configuration and no Rust knowledge, whereas the Modular SDK, while powerful, still requires some coding in Rust. Both proposed solutions support Docker-based deployment and seamless integration with Supabase, enabling rapid setup and scalable storage. The native Sui approach lacks these modern DevOps features. Finally, both the Modular SDK and Generic Indexer support real-time data ingestion, whereas the native approach is limited by RPC latency and data pruning.

5.4 Summary and Comparison

Our experimental evaluation confirms that both the Modular SDK and Generic Indexer dramatically reduce the time and complexity required for Sui blockchain data indexing. The Modular SDK offers the greatest flexibility and speed, while the Generic Indexer provides a zero-code, rapid deployment path for non-technical teams. Both solutions outperform the native Sui approach in speed (up to 52× faster), reduce query complexity from $O(NQ)$ to $O(Q)$, and enable full data self-custody.

6 Conclusion and Perspectives

6.1 Summary of Contributions

This work addressed a central challenge in the Sui blockchain ecosystem: the high cost and complexity of on-chain data indexing. Our research confirmed that development teams spend a significant portion of their project time, up to 30%, on building and maintaining indexing solutions, which diverts resources from core application logic and slows time-to-market. To address this, we introduced two complementary contributions: a *Modular SDK* for fine-grained, customizable indexing, and a *Generic Package-Based Indexer* for rapid, zero-code deployment.

The Modular SDK empowers developers to specify exactly which transaction fields to process and store, and to define custom Rust callbacks for data transformation. This approach enables selective, efficient indexing tailored to each application's needs, while ensuring that all data remains under the team's control. The SDK is built on top of the Sui Alt Framework and integrates seamlessly with PostgreSQL and Supabase, providing a modern, developer-friendly workflow.

The Generic Indexer, by contrast, is designed for accessibility and ease of use. It automatically ingests all transactions related to a specified package, with minimal configuration required. This solution is ideal for teams without deep Rust expertise or those seeking a fast path to data access and analytics. The indexer is fully containerized, supports both local and remote modes, and outputs data in a format ready for integration with visualization tools such as Appsmith or Dune.

6.2 Limitations

Despite these advances, several limitations remain. The Modular SDK, while flexible, still requires developers to write Rust code for custom callbacks and to manually define data types that mirror Move structures or database schemas. This introduces boilerplate and can slow onboarding for teams less familiar with Rust. Automatic type derivation from Move contracts or database schemas is not yet implemented, and remains an important area for future work. Generative model could fill that gap.

The Generic Indexer, while easy to deploy, sacrifices flexibility for simplicity. It indexes all data related to a package, which can result in over-collection and increased storage requirements, especially for high-traffic packages. The lack of in-pipeline filtering or custom logic means that some use cases may require a second analytical pass, increasing compute effort.

6.3 Perspectives and Future Work

Future development will focus on several key areas.
First, automating type derivation from Move contracts and database schemas will further reduce manual effort and make the Modular SDK accessible to a broader range of developers.
Second, introducing adaptive filtering and incremental processing in the Generic Indexer will help avoid over-indexing and improve efficiency for high-traffic packages like Dex packages. Finally, deeper integration with visualization platforms like Dune as well as support for streaming outputs to real-time dashboards will further streamline the analytics pipeline.

6.4 Final Remarks

In summary, this work demonstrates that efficient, self-hosted indexing is both practical and highly beneficial for Sui application development, especially in early stages. By lowering the technical barrier to entry and enabling full data self-custody, our solutions allow teams to focus on building innovative applications rather than reinventing data infrastructure. While some limitations remain, the results confirm that a significant share of the development effort traditionally spent on data engineering can be eliminated.

Acknowledgments

We would like to express our sincere gratitude to Gauthier Voron for his guidance and support as our supervisor throughout this project. We are also deeply thankful to Professor Rachid Guerraoui for accepting our project within the DCL laboratory at EPFL, providing us with the opportunity to develop our work in a stimulating academic environment.

Special thanks go to the Developer Relations team at Sui in Athens, whose insightful feedback helped refine our project's direction and focus. We are also grateful to Ashok, who is in charge of the indexing component at Mysten Lab. Meeting him in person and receiving his constructive feedback was very helpful to our progress.

Definition of Terms

Blockchain A distributed ledger technology that maintains a continuously growing list of records, called blocks, which are linked using cryptography.

Checkpoint In Sui, a checkpoint is a collection of transaction certificates that have been finalized. Checkpoints serve as the unit of finality in Sui's consensus protocol.

dApp Decentralized Application. A computer application that runs on a distributed computing system rather than a single computer.

Data Indexing The process of organizing and structuring data from a blockchain to make it more accessible, queryable, and useful for applications.

Data Self-Custody The principle of maintaining full ownership and control over one's data, rather than relying on third-party services to store or manage it.

DeFi Decentralized Finance. A blockchain-based form of finance that doesn't rely on central financial intermediaries.

Docker A platform used to develop, ship, and run applications inside containers, ensuring consistent behavior across different environments.

Full Node A network node that maintains a complete copy of the blockchain, including all transactions and state transitions.

Indexer A system that processes blockchain data and organizes it into a database structure that can be efficiently queried.

JSON JavaScript Object Notation. A lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.

Move A programming language designed for writing safe smart contracts, originally developed for the Diem blockchain and adopted by Sui.

Object-Centric Model Sui's approach to data representation, where the fundamental unit of storage is an object with a unique identifier, rather than an account.

Package In Sui, a package is a collection of Move modules published together. It's the unit of smart contract deployment.

PostgreSQL An open-source relational database management system emphasizing extensibility and SQL compliance.

Rust A multi-paradigm, high-level, general-purpose programming language designed for performance and safety, especially safe concurrency.

SDK Software Development Kit. A collection of software tools, libraries, and documentation that helps developers create applications for a specific platform.

Smart Contract Self-executing contracts with the terms directly written into code. They automatically execute transactions when predetermined conditions are met.

Sui A Layer 1 blockchain designed for high throughput and low latency, featuring an object-centric data model and the Move programming language.

Transaction An atomic operation that changes the state of the blockchain, such as transferring assets, creating objects, or executing functions on smart contracts.

Transaction Digest A unique identifier for a transaction, typically represented as a cryptographic hash of the transaction data.

Bibliography

References

[1] Mysten Labs, "Sui: A Next-Generation Smart Contract Platform with High Throughput, Low Latency, and an Asset-Oriented Programming Model," Whitepaper, 2022.

[2] Mysten Labs, "Sui Developer Documentation," https://docs.sui.io/, 2023.

[3] Sentio, "Sentio Documentation: Sui Indexing," https://docs.sentio.xyz/, 2023.

[4] Mysten Labs, "Sui Alt Framework," GitHub Repository, https://github.com/MystenLabs/sui, 2023.

[5] Diesel, "Diesel: A Safe, Extensible ORM and Query Builder for Rust," https://diesel.rs/, 2023.

[6] Alexandre Mourot, Loris Tran, "Obsuidian Github Repository," https://github.com/Project-Magma-Monorepo/Obsuidian, 2025.

[7] Loris Tran, "Obsuidian: a fully decentralized indexer solution for the Sui network," https://github.com/Project-Magma-Monorepo/Obsuidian/blob/main/Master_Project_Obsidian_Loris___Alexandre.pdf, 2025.

[8] Supabase, "Supabase: An open source Firebase Alternative," https://supabase.com/docs, 2024.