SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics in Informatics

Results on Results: Building new results from cached partial results

Marcio Brito Barbosa

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Data Engineering and Analytics in Informatics

Results on Results: Building new results from cached partial results

Results on Results: Erstellung neuer Resultate aus zwischengespeicherten partiellen Resultaten

Author:	Marcio Brito Barbosa
Examiner:	Prof. Dr. Thomas Neumann
Supervisor:	Prof. Dr. Peter Boncz, Boaz Leskes (MotherDuck)
Submission Date:	25.08.2025

I confirm that this master's thesis in data engineering and analytics is my own work and I have documented all sources and material used.

Munich, 25.08.2025 Marcio Brito Barbosa

Acknowledgments

The research for this thesis was conducted at the MotherDuck office in Amsterdam, and I am grateful to the entire company for the opportunity and resources that made this work possible. The innovative and collaborative environment at MotherDuck provided the direct inspiration and motivation for this research.

I would like to extend my gratitude to my advisors, Peter Boncz and Boaz Leskes, whose insightful guidance was instrumental in shaping the direction of this thesis.

Abstract

The interactive experience of writing Structured Query Language (SQL) has not kept up with the performance advancements of modern analytic engines. In particular, analytical cloud services almost always incur Wide Area Network (WAN) latency, which means that even short queries can feel slow to analysts. As a result, users are often left in a "write–run–wait" cycle that hampers exploratory analysis. Newer features such as MotherDuck's Instant SQL [25c] seek to offer analysts a fluid exploratory experience by providing real-time query previews. Powering this capability in a modern hybrid architecture requires a new paradigm of caching.

A promising approach for these interactive functionalities is to work with partial, client-side caches — which we call **Results** [NA23]. The primary contribution of this dissertation is to go beyond basic caching to the composition of **Results**, treating query previews as a form of Approximate Query Processing (AQP). However, this powerful technique exposes a critical new problem that threatens the viability of the method: **insufficient result cardinality** when results are computed on previous data samples. Composing partial data samples can produce a result set that is too small to be useful, compromising the interactive user experience.

This thesis develops and evaluates an intelligent recovery framework to efficiently resolve these failures. Rather than recompute the entire resolved query on the server — a potentially expensive operation — the framework deploys algorithms to intelligently select and re-create a single upstream dependency, representing a far more efficient approach. We develop and assess three novel **heuristics** to guide the selection of the upstream dependency: one via optimizer Cardinality Estimation, another via empirical Dynamic Sampling, and a third via analytical Data Lineage.

These strategies were benchmarked in a simulated environment with realistic, multi-step analytical workloads generated by Large Language Models (LLMs) fed with both the standard Transaction Processing Performance Council - H (TPC-H) and the skewed Join Crossing Correlations - H (JCC-H) data. The experimental results demonstrate that **Data Lineage** offers the best overall performance with reliable predictions, the best response time, and the fewest number of user interruptions, with virtually no additional computational cost for the client. In contrast, the optimizer's cardinality estimates proved highly unreliable, and the exhaustive sampling method was found to be unfeasible due to its high local cost. This work demonstrates that it is possible to provide a lightweight analytical heuristic based on data provenance to manage failures caused by insufficient result cardinality, making real-time query previews a practical reality in modern hybrid database systems.

Contents

1 Introduction

1.1 Background and Foundational Concepts

This chapter provides the necessary background in database systems to understand the context and contribution of this thesis. We begin with the foundational relational model and query languages, then delve into the internal mechanics of query processing and optimization, and conclude with the advanced techniques that directly inform the strategies developed in this work.

1.1.1 The Relational Model and SQL

The relational data model is both the theoretical and practical basis for the vast majority of modern database systems. The model was first proposed by Edgar F. Codd in 1970, and while other data models have appeared, the principles established in this model have shown remarkable longevity and have become dominant in the field [Cod70]. The central abstraction in the model is the relation, which is physically represented as a table. Data are organized into these two-dimensional structures, which are composed of rows, formally known as tuples, and columns, or attributes. Each tuple represents one related set of data points, while each attribute is defined over a domain of allowable values. The relational model is particularly useful because it allows for the linkage of distinct relations to one another based on common key values that establish logical connections between different datasets. Thus, a relational database is defined as an organized collection of interrelated relations [WE14a].

SQL was developed as the method of interacting with and manipulating data within this model. SQL was first developed by IBM in the early 1970s under the name SEQUEL and then further commercialized; it has now evolved into the common language for relational database management systems [WE14b].

SQL is characterized as a high-level, declarative language. This allows users to specify the intended effect when conducting an operation against the data contained within this model, without specifying how the system should implement that operation. For example, an operation that finds the names of all patients over 65 years of age could be expressed as follows:

```
SELECT Name
```

```
FROM Patient
WHERE Age > 65;
```

This statement indicates what data are needed, but the details of implementation (such as methods for scanning tables, use of indexes, or join algorithms) are left to the database's query optimizer. In short, SQL provides an important bridge between the user and the abstract constructs of the relational model, allowing users to create complex queries against table-structured data.

1.1.2 Relational Algebra and Query Trees

While SQL provides a declarative means of querying a database, relational algebra is the theoretical, procedural basis for how those queries are executed. Relational algebra (also defined by Codd) is a formal language that uses a finite number of known operators to manipulate relations [Cod70]. Each operator takes one (or more) relations as input and outputs another relation. This algebra provides a formal way of specifying non-trivial operations for retrieving data. There are a number of operators that make up the algebra, and we can classify these as unary (one relation) or binary (two relations) operators. The main primitive operators are as follows:

- **Selection (σ):** Filters tuples based on a specified predicate or condition. It returns a horizontal subset of the input relation containing only the tuples that satisfy the condition.
- **Projection (π):** Selects a subset of attributes from a relation, discarding the others. It produces a vertical subset of the input relation.
- **Union ($\cup$):** Computes the set union of two union-compatible relations (i.e., having the same number and type of attributes), returning all unique tuples present in either relation.
- **Set Difference ($-$):** Returns all tuples that are present in the first of two union-compatible relations but not in the second.
- **Cartesian Product ($\times$):** Generates a new relation composed of all possible combinations of tuples from two input relations. If relation R has n tuples and relation S has m tuples, their Cartesian product $R \times S$ contains $n \cdot m$ tuples.
- **Rename (ρ):** Changes the name of a relation or its attributes, which is crucial for resolving ambiguity, particularly in self-joins or complex products.

Many other operators can be defined with these fundamental operators, most significantly the **Join ($\bowtie$)** operator. A natural join, for example, is a composite operator consisting of a Cartesian product, selection, and projection that take tuples from two relations and combine them in accordance with the equality of their shared attributes.

The logical representation of a query is often expressed in the form of a query tree. In such tree data structures, the leaves represent base relations found in the database to which the query relates, and the internal nodes represent the various relational algebra operations applied to combine those leaves. The data flow proceeds from the leaves to the root, which returns the logical result for the query.

The query tree is a clear, manipulable representation of the steps necessary to compute the query. This is important to the process of query optimization. A query optimizer in a database can apply heuristic and/or cost-based rules to transform an initial query tree into a logically equivalent but more efficient one. For example, a query tree can be optimized by "pushing" selections and projections as low as possible to minimize intermediate result sizes, while reordering joins to reduce computation costs. This logical plan is then converted to a physical execution plan for the query engine. This shows that relational algebra and query trees can provide theoretical and structural understanding of the process by which a declarative SQL query is transformed into an efficient, procedural execution plan.

1.1.3 The Query Processing Pipeline

The life of a query in a contemporary Relational Database Management System (RDBMS) is defined by a series of steps performed in a formal pipeline structure. This process is primarily responsible for transforming a high-level declarative SQL statement into a low-level procedural execution plan that the system can run efficiently. Typically, the architectural design of this pipeline, building on the foundational work of Selinger et al. and Graefe, consists of four phases: parsing and semantic analysis, logical optimization (rewriting), physical (cost-based) optimization, and execution [Sel+79; Gra93].

1. **Parsing and Semantic Analysis:** The execution of a query begins once the Database Management System (DBMS) receives a string that represents an SQL query. The parser first checks the statement for syntactic correctness according to the SQL grammar and converts it into an internal representation, typically a parse tree. Once the parse phase has been completed, the semantic analysis stage occurs, where the parser verifies the query against the DBMS's system catalog (or metadata). Specifically, the parser checks whether all relation and attribute references are defined and whether attributes are type-compatible in

predicates and expressions. The parser may also resolve any referenced views to their definitions. The output of this phase is a valid initial logical query plan that can be represented as a tree of relational algebra operators.

2. **Logical Optimization (Query Rewriting):** The initial logical plan of the query may not always be optimal. The next phase is a series of heuristic, rule-based transformations that put the plan into a logically equivalent, but more efficient form. The transformations are based on the mathematical equivalences of relational algebra. They include common rewriting rules such as "pushing down" selection predicates as far down in the plan as possible in order to reduce the volume of intermediate results, merging multiple operations together, and removing unnecessary sub-expressions. Logical optimization may also involve reordering operators, such as joins, based on logical cost metrics like the estimated number of tuples produced by a query subtree. This stage of query processing operates at the logical level. It will only change the structure of the plan, but it does not yet define the physical costs to perform any of the physical operations.

3. **Physical (Cost-Based) Optimization:** Physical optimization focuses on choosing among multiple physical algorithms that can implement a given relational operator. The optimizer starts with the rewritten logical plan and explores a large space of potential physical execution plans. For each logical operator in the plan (i.e., join, sort, scan), it considers alternative physical algorithms (i.e., hash join vs. nested-loop join; filesort vs. indexed retrieval). Using a cost model and statistics about the data stored in the database, it estimates the resource cost (usually a function of I/O and CPU) of running each physical plan. The goal is to select the plan with the lowest estimated cost.

4. **Execution:** Once the optimal physical plan has been determined, the execution engine of the database receives the plan. The execution engine takes the plan, which is a sequence of specific physical operators, and executes the plan one operator at a time. As a result, the data specified in the query are retrieved, and the final result set is produced and returned to the user.

In conclusion, this pipeline is a systematic decomposition of the query processing problem, from a high-level declarative specification to a more optimized, low-level execution strategy. Understanding this process is essential to analyzing database performance and the behavior of complex queries.

1.1.4 Fundamentals of Cardinality Estimation

Reliable cardinality estimation is a fundamental requirement for any cost-based query optimizer. The optimizer must decide which of a potentially very large set of alternatives is likely to produce the most efficient execution plan. These cost estimates depend critically on the estimated number of tuples (i.e., cardinality) produced by the intermediate operators in the query plan [Lei+15]. If the optimizer has inaccurate cardinality estimates, it might select a seriously inefficient plan; for example, it could pick the worst join ordering or even the wrong access method, degrading query performance by many orders of magnitude [IC91; Poo+96].

In order to produce cardinality estimates, the optimizer must rely on statistical metadata—also known as synopses—that summarize the data distributions of the base relations. The metadata are precomputed from the base relations and stored in the system catalog of the database. There are many types of synopses, but the most common are:

- **Histograms:** These structures approximate the distribution of values within a single column by partitioning the values into a set of buckets and storing frequency counts for each. They are particularly useful for estimating the selectivity of range predicates.
- **Number of Distinct Values (NDV):** A count of the unique values in a column, essential for estimating the result size of grouping operations and joins. In practice, NDV is often estimated using a HyperLogLog data structure, which is compact and easy to maintain.
- **Minimum and Maximum Values:** These simple statistics allow the optimizer to quickly determine whether a range predicate falls outside the bounds of the data in a column. They are also used to skip reading parts of the table (such as partitions or row groups), which can significantly reduce the number of tuples that need to be processed.
- **Most Frequent Values (MFVs):** A list that stores the "heavy hitter" values and their exact frequencies. This helps correct for data skew, where a small number of values appear much more often than others.

These values are gathered through background processes executed periodically, or explicitly triggered by a Database Administrator (DBA) (e.g., using an `ANALYZE` command).

Even when estimates are generated from statistics, consistently estimating cardinality is notoriously difficult because the probabilities are based on several simplifying assumptions that are often incorrect in real-world data.

One of the major assumptions made is that the attribute values are independent. Consider the case of a `cars` table: predicates such as `model='ID.4'` and `make='VW'` are clearly correlated, but a simple estimator may multiply their selectivities as if they were independent, yielding very inaccurate probabilities about the result.

Ioannidis and Christodoulakis demonstrated how even very small estimation errors from base relations get multiplied together exponentially as they move upward through a query tree [IC91]. This is why continuing work on improving cardinality estimation methods and metrics will always be necessary, and remains an important area of research within databases.

1.1.5 Sampling-Based Cardinality Estimation

Synopsis-based methods using histograms and precomputed statistics form the cornerstone of traditional query optimization; however, due to skewed data, correlated attributes, and dynamic distributions, they can be considerably limited. A second complementary approach is sampling-based estimation. Instead of only relying on stored metadata, sampling-based estimation uses data to conduct statistical inferences. The fundamental principle is to execute a query on a small, randomly selected sample of the data and then extrapolate the observed cardinality to the full dataset.

This approach has been influential in the development of systems for online and interactive analytics. An example is the work on `Ripple Joins`, which process joins in a sampling-based, incremental fashion to provide users with progressively refined estimates and confidence intervals [HH99]. More recently, the principles of sampling have become central to the field of AQP. This can be seen in the systems `BlinkDB` [Aga+13] and `VerdictDB` [Par+18b], which leverage precomputed data samples (stratified or uniform) to provide responses to analytical queries with very low latency, producing estimated results with defined bounds for potential error. Although the premise behind AQP systems is often focused on providing data analysts with fast approximate answers, the sampling-based estimation metrics defined in AQP are also applicable to providing cardinality estimates for cost-based query optimization.

The primary strength of sampling-based systems is their resilience to complex relationships in the data. By using a direct sample of the data, they can naturally capture the effects of attribute correlation and value skew without the need for complex multidimensional synopses or the restrictive assumptions of attribute independence. This enables sampling-based systems to react to potentially dynamic environments, where the data distributions could shift drastically.

However, sampling entails some inherent trade-offs:

- **Runtime Overhead:** Sampling requires accessing data at query time, which incurs

I/O and CPU overhead. This cost must be balanced against the potential benefit of a more accurate plan.

- **Estimation Variance:** The accuracy of a sampling-based estimate is a function of the sample size. For queries with very high selectivity (i.e., those that match very few tuples), a small random sample may fail to capture any qualifying tuples, leading to a cardinality estimate of zero and high statistical variance.

Despite these challenges, sampling-based estimation is a powerful technique in the query processing landscape. It naturally captures correlations: for instance, if the optimizer evaluates a predicate such as `model='ID.4'` and `make='VW'` on the sample, the resulting estimate will correctly reflect the effect of the correlation. Sampling-based estimation now serves as a tool for modern analytical databases, distributed query engines, and systems where providing fast, approximate results is a key requirement. While in the early days predicate evaluation on samples was considered too costly—since CPUs were slow and even scanning a few thousand tuples added overhead—on current hardware this evaluation is fast enough to be practical during optimization.

1.1.6 Client-Side Caching of Query Results

In client-server database architectures—common in cloud-based computing—a considerable aspect of query latency is attributable to network latency and server-side load when generating data. One approach to reducing server-side load, and consequently data retrieval latency, is to cache the results of previous queries on the client side. When a new query is received, the first evaluation or verification for the system is whether these results can be partially or fully synthesized from the previously cached data. If the previous query results indicate that there is a partial or full answer that can subsequently be served from the client-side cached data, it is beneficial to avoid the round trip to the server to obtain results from the database, thus significantly reducing the time to return a result set and decreasing the server infrastructure workload needed to retrieve the data [KB96].

Implementing client-side result caching introduces two fundamental challenges: cache completeness and cache consistency. Cache completeness relates to the problem of deciding whether the local cache contains all the necessary data to answer a new query without contacting the server. This often requires metadata that explain the contents of the original query, e.g., the predicates present in the queries that populated the cache [KB96]. Cache consistency, on the other hand, is the challenge of ensuring that the data stored in the cache remain synchronized with the master data on the server. Cached results could become stale if the data in the source server are modified, resulting in cache invalidation and/or update propagation scheme requirements.

Studies in this domain have extended the focus of caching not only to final query results, but also to intermediate results [Iva+09]. By caching intermediate results produced in the query plan of a query, the system can potentially take advantage of those intermediate results to serve queries in the future, building more efficient execution plans. Caching intermediate results presents the problem of co-locating the caching subsystem with the query optimizer, leaving the optimizer in a position to make cost-based decisions on what to cache and how to take advantage of cached data when creating new execution plans [Roy+00]. While the details of caching methods are beyond the scope of this background, the reuse of previous computations to accelerate the time for subsequent queries is a central theme.

1.1.7 Approximate Query Processing

Building on the idea of reusing data in client-side caching, AQP provides a more structured way to obtain fast answers from large datasets. Whereas a client's cache can be viewed as an ad hoc collection of data from past queries, AQP is intentional. It formalizes the notion of representing the whole with a smaller portion by operating on carefully designed summaries or statistical samples. The aim of AQP is straightforward: give up perfect accuracy to achieve a large gain in query speed [HHW97].

AQP systems typically rely on precomputed, offline samples of the data; for instance, systems such as `BlinkDB` and `VerdictDB` build small stratified samples intended to approximate the entire dataset [Aga+13; Par+18a]. When a user issues a query (for example, `SELECT COUNT(*)` or `SELECT AVG(col)`), the system evaluates it against the sample rather than the full dataset. From the sample statistics, the system derives an approximate answer and attaches a confidence interval. The user therefore gains both a quick response and an indication of the possible error, which is often sufficient for exploratory analysis and identifying trends.

This approach, however, introduces challenges distinct from cache management. The guarantees of AQP are strongest for simple aggregate queries; accuracy is harder to ensure when queries involve complex operations such as `JOIN`s or subqueries. In addition, if the underlying data are highly non-uniform—where some values appear disproportionately often—or if the query targets a rare event, then even a well-chosen sample may fail to be representative, leading to poor estimates. Addressing such cases and extending AQP to more complex queries and distributions remains an active line of database research.

1.1.8 Lineage and Data Provenance

Although the previous chapters focused on query execution efficiency—whether in terms of cost-based optimization or approximation—we now consider the semantics of what a query's result means. Data provenance, often used interchangeably with data lineage, is the study of tracing data back to its origins and transformations. For any particular query, the provenance of an output tuple is the set of input tuples from the source tables that were responsible for its creation. In other words, it answers the question: "Which pieces of the input data were used to generate this piece of the output?"

A seminal paper in this line of work by Buneman et al. formally characterized two broad kinds of provenance: why-provenance and where-provenance [BKT01]. Why-provenance refers to which input tuples caused an output tuple to exist—for example, in a `JOIN` operation it would identify the pair of source tuples from the input tables that matched to produce a result. Where-provenance, on the other hand, concerns the physical locations (e.g., table and column names) from which the data values in an output tuple were copied.

There are several reasons why understanding data provenance is important in modern database systems. It provides a foundation for:

- **Debugging and Explanation:** When a developer encounters unexpected or incorrect results from a query, provenance makes it possible to trace the problem back to the specific input data that caused it [DG15].
- **Data Quality and Trust:** With an explicit audit trail of how the data was generated, provenance supports assessing the trustworthiness and quality of the data used in analytics and reporting [Dai+08].
- **Scientific Reproducibility:** In scientific settings, provenance not only documents the original data and processes used in an experiment but also enables others to validate and reproduce the research [Cra+15].

A full treatment of provenance-capture techniques lies beyond the scope of this background. The key idea, however, is that query outputs can be mapped back to the specific inputs that produced them—an idea central to the present thesis.

1.1.9 DuckDB and the Rise of Hybrid Execution with MotherDuck

DuckDB is an embeddable analytical database management system designed to run analytical SQL queries while embedded in another process. It was created to address the gap left by `SQLite`. While `SQLite` excels in transactional workloads, it suffers from inefficiencies in analytical processing. DuckDB, in contrast, is tailored for Online Analytical Processing (OLAP) queries and is well suited to data science and analytics workflows where analytical processes must handle large datasets efficiently [RM19].

Key features of DuckDB include:

- **High Performance for Analytical Workloads:** `DuckDB` uses a vectorized execution engine based on the MonetDB/X100 "Hyper-Pipelining Query Execution" approach [BZN05], and processes data in batches, improving performance for analytical queries compared to traditional row-based systems.
- **Embeddability:** `DuckDB` runs as part of the host process, meaning there is no standalone database server to maintain. This design simplifies deployment and avoids the overhead of transferring data to a separate server, a concern also highlighted in the "Don't Hold My Data Hostage" paper by Hellerstein and Stonebraker [RM17].
- **Portability:** DuckDB is written in C++ and has no external dependencies, simplifying integration across environments.
- **Open Source:** DuckDB is released under the MIT license, encouraging community contributions and broad accessibility [25a].

DuckDB is built for analytical workloads and includes components such as a parser, logical planner, optimizer, physical planner, execution engine, concurrency control, and storage manager. The parser uses a modified version of PostgreSQL's SQL parser for robust query parsing. The optimizer is cost-based, and the storage implementation uses a read-optimized columnar format with lightweight indexing to reduce unnecessary data reads. DuckDB supports a wide range of SQL capabilities and effectively works with large datasets in memory, even on devices with limited resources—making it well suited for scenarios such as interactive data analysis and edge computing [RM19].

1.1.10 MotherDuck

`MotherDuck` is a serverless analytics platform built on top of `DuckDB`. Unlike traditional systems that rely on "scale-out" clusters, it takes a "scale-up" approach, running most analytical workloads on a single machine. The idea is that many real-world analytical tasks are not as large as people assume and can be handled more effectively on a single node [25d].

What makes `MotherDuck` distinctive is its dual, or hybrid, execution model. Queries can be split so that some parts run on a local client machine while others are executed in the cloud. This setup makes it possible to balance performance, cost, and data locality by moving the work to wherever the data happens to be.

Query Shipping vs. Data Shipping

Distributed database systems have long debated two strategies for moving queries and data:

- **Query Shipping:** The query itself is sent to the server that holds the data. This works well for large datasets and relatively simple queries, since it avoids unnecessary data transfer.
- **Data Shipping:** The data is sent to the location where the query is running. This can be useful when queries are more complex or when the dataset is small enough to move without much overhead.

`MotherDuck` combines the two. By splitting a query plan between client and cloud, it can decide case by case which operations should stay local and which are better executed remotely [25d].

Dual (Hybrid) Execution

`MotherDuck`'s dual execution framework combines the fast, in-process analytical engine of `DuckDB` with the flexibility of a cloud service. When a user connects a local `DuckDB` instance to `MotherDuck` (for example, using the command `ATTACH 'md:';`), the local process effectively becomes a node in the larger `MotherDuck` ecosystem. Through this connection, the local database can reach cloud-persisted data and take part in hybrid query planning [25d].

Planning and executing a hybrid query involves a few main steps:

1. **Query Decomposition:** The query is parsed into a directed acyclic graph (DAG) of relational operators.
2. **Cost Estimation and Site Selection:** The optimizer decides, based on data locality, whether each operator should run locally or remotely.
3. **Bridge Operator Insertion:** Special "bridge" operators are placed in the plan to move data between local and remote parts.
4. **Result Materialization:** The final result is produced in the most efficient location, minimizing the amount of data transferred back to the user.

This framework supports different execution strategies—fully local, fully remote, or hybrid—making it possible to build low-latency interactive applications that take advantage of both local resources and the cloud [25d].

WebAssembly (WASM)

WebAssembly (WASM) is a binary instruction format that serves as a compilation target for languages such as C++. The main idea is to let high-level languages run inside a web browser at speeds close to native. It does this by providing a compact, safe, and efficient standard for executing code in the browser environment [25e].

For `DuckDB` and `MotherDuck`, WASM plays an important role. It makes it possible to run `DuckDB` directly in the browser, which is especially useful when building interactive applications that demand very low latency. With this setup, queries can be issued against data that lives locally or in the cloud. By compiling `DuckDB` to WASM, users of `MotherDuck` gain the ability to run SQL queries in the browser itself, while the hybrid execution model decides what should be handled locally and what should be pushed to the cloud.

1.1.11 Conclusion of Background

This chapter has outlined the core background needed for the rest of the thesis. We began with the fundamentals of the relational model and SQL, then moved through query processing and optimization, and finally looked at more advanced topics such as cardinality estimation, approximate query processing, and the hybrid architecture behind `MotherDuck`. Together, these ideas provide the context for the contributions that follow.

2 Problem Statement

This chapter establishes the foundational context for this thesis. We begin by exploring the architectural shifts in modern data analysis that motivate this work. We then formalize the definition and mechanics of `Results`, a key building block. Finally, we articulate the novel research problem that this thesis aims to solve: the composition of `Results`.

2.1 The Rise of Interactive Data Applications

Modern data analysis increasingly relies on interactive data applications that provide immediate insights through visualizations and dashboards. A key technology in this space is **DuckDB**, a high-performance analytical database engine that can run directly within a web browser's WebAssembly (Wasm) environment. This capability allows for complex analytical queries to be executed on the client device, enabling highly responsive applications by eliminating network latency.

MotherDuck extends this paradigm into a serverless data warehouse by pioneering a **hybrid query execution** model [AA23]. This model intelligently splits the processing of a single query between the user's local device and the powerful cloud backend. The decision of where to execute specific query fragments depends on the location of the data.

However, this powerful architecture presents two primary challenges:

1. **Network Latency:** Transferring data or query results between the MotherDuck cloud and the browser introduces significant latency, which can undermine the interactive feel of an application.

2. **Data Locality Requirement:** The benefits of local processing are only realized when data is present on the client device. If all data resides in the cloud, the user's local compute resources remain idle, and every query incurs a network roundtrip.

Addressing these issues is paramount for creating truly fluid and powerful interactive data applications.

2.1.1 Motivation: The Instant SQL Feature as a Motivating Application

Modern data analysis has seen tremendous performance gains in query engines, yet the interactive experience of writing SQL has remained largely static. Analysts are often trapped in a tedious “write–run–wait” cycle, partly due to the Wide Area Network (WAN) latency inherent in cloud-based systems, which disrupts the cognitive flow essential for deep data exploration. Despite the ability of engines like DuckDB to scan billions of rows in seconds, the user experience is still gated by the latency between authoring a query and observing its result.

To address this gap, MotherDuck developed **Instant SQL** [25c], a feature integrated into its local UI that fundamentally changes the nature of query writing. This section describes the Instant SQL feature and establishes it as the primary motivation for the caching and query composition mechanisms explored in this thesis.

The MotherDuck UI and the Instant SQL Experience

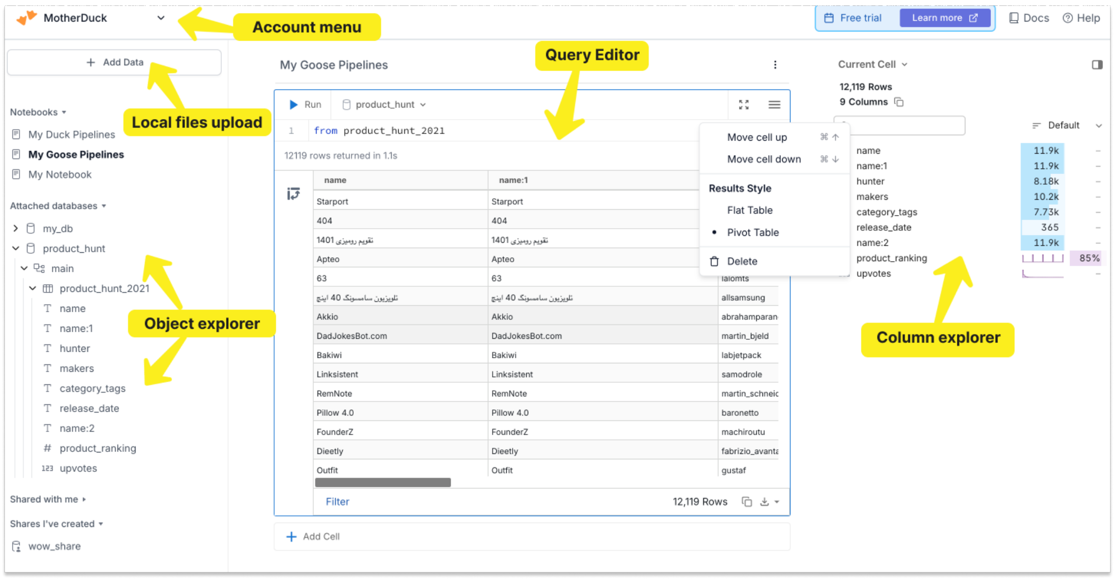
The MotherDuck UI is a local, notebook-style SQL environment designed for interactive data analysis. It allows users to connect to and explore data from various sources simultaneously, including local files, remote object stores, and databases. Its interface provides tools like a data catalog and a table explorer that offers immediate visual diagnostics, such as column distributions and null percentages, without requiring users to write boilerplate queries.

The flagship feature of this UI is **Instant SQL**. Instead of the traditional "write-run-wait" paradigm, Instant SQL provides a real-time preview of a query’s result set that updates as the user types. This transforms the static query editor into a live, interactive canvas. An analyst can write a complex query with multiple Common Table Expressions (CTEs) and see the output of each intermediate step instantly. This immediate feedback loop makes it effortless to debug complex expressions, experiment with different analytical approaches, and maintain a state of analytical flow.

When satisfied with the real-time preview, the user can then choose to run the query to materialize the final, complete result.

It is important to emphasize that **Instant SQL is a preview and exploration tool, not a replacement for running the full query**. Because it operates on partial, cached samples of the data, the previews it generates are approximations. While excellent for checking the logic of transformations and joins, they may not be numerically accurate, especially for aggregate functions like `COUNT(*)` or `SUM()`. This is an intentional design trade-off, prioritizing immediate feedback and workflow fluidity over the guaranteed correctness required for final reporting.

This allows the analyst to perfect the query’s logic using the instant preview. Once



Notebooks
My Duck Pipelines
My Goose Pipelines
My Notebook

Attached databases
my_db
product_hunt
main
product_hunt_2021
name
name:1
hunter
makers
category_tags
release_date
name:2
product_ranking
upvotes

Shared with me
Shares I've created
wow_share

My Goose Pipelines

product_hunt

```
from product_hunt_2021
```

12119 rows returned in 1.1s

name	name:1	
Starport	Starport	
404	404	
تقویم رومیزی 1401	تقویم رومیزی 1401	
Apteo	Apteo	
63	63	lalomts
تلویزیون سامسونگ 40 اینچ	تلویزیون سامسونگ 40 اینچ	allsamsung
Akkio	Akkio	abrahamparan
DadJokesBot.com	DadJokesBot.com	martin_bjeld
Bakiwi	Bakiwi	labjetpack
Linksistent	Linksistent	samodrole
RemNote	RemNote	martin_schneic
Pillow 4.0	Pillow 4.0	baronetto
FounderZ	FounderZ	machiroutu
Dieetly	Dieetly	fabrizio_avanta
Outfit	Outfit	gustaf

12,119 Rows

Move cell up ⌘ ↑
Move cell down ⌘ ↓
Results Style
Flat Table
• Pivot Table
Delete

Current Cell

12,119 Rows
9 Columns

Default

name	11.9k	–
name:1	11.9k	–
hunter	8.18k	–
makers	10.2k	–
category_tags	7.73k	–
release_date	365	–
name:2	11.9k	–
product_ranking		85%
upvotes		–

Figure 2.1: An overview of the MotherDuck local UI, showing the SQL notebook, data catalog, and table explorer panels.

the logic is confirmed, they then execute the full query against the complete dataset to obtain the final, accurate results.

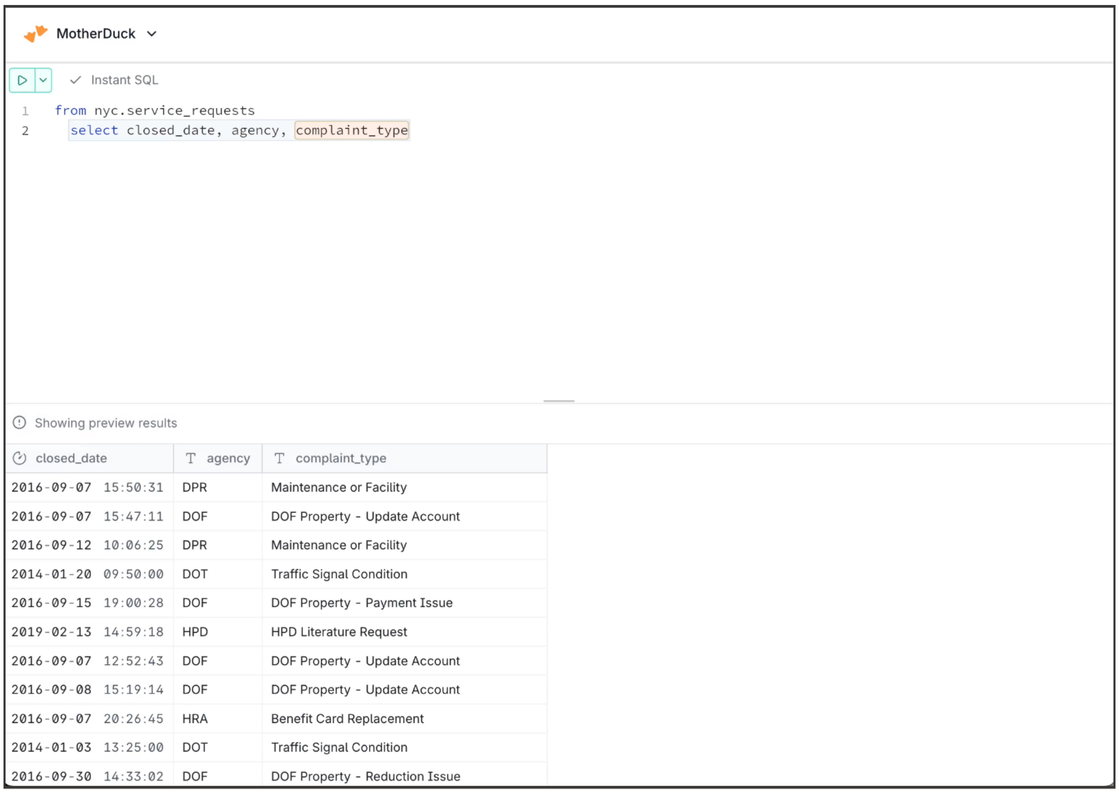
The Technical Challenge: Enabling Instantaneous Previews

Achieving this instant feedback loop presents a significant technical challenge. While DuckDB is exceptionally fast, many real-world queries—especially those involving joins, aggregations, or remote data sources—will take longer than the roughly 100ms threshold for human perception of "instantaneous." Naively re-executing the full query on every keystroke is therefore infeasible.

The solution requires a more sophisticated architecture capable of:

1. **Local Data Caching:** Intelligently caching partial samples of results.
2. **Dynamic Query Rewriting:** Programmatically parsing the user's active query text and rewriting it on-the-fly to point to these local, partial caches to generate a preview.

The `Result` operator, which forms the basis of this thesis, was developed as the foundational technology to solve this exact problem. While the current implementation of Instant SQL relies on a more direct approach of caching data in temporary tables,



```
from nyc.service_requests
  select closed_date, agency, complaint_type
```

Showing preview results

closed_date	agency	complaint_type
2016-09-07 15:50:31	DPR	Maintenance or Facility
2016-09-07 15:47:11	DOF	DOF Property - Update Account
2016-09-12 10:06:25	DPR	Maintenance or Facility
2014-01-20 09:50:00	DOT	Traffic Signal Condition
2016-09-15 19:00:28	DOF	DOF Property - Payment Issue
2019-02-13 14:59:18	HPD	HPD Literature Request
2016-09-07 12:52:43	DOF	DOF Property - Update Account
2016-09-08 15:19:14	DOF	DOF Property - Update Account
2016-09-07 20:26:45	HRA	Benefit Card Replacement
2014-01-03 13:25:00	DOT	Traffic Signal Condition
2016-09-30 14:33:02	DOF	DOF Property - Reduction Issue

Figure 2.2: The Instant SQL feature in action, providing a real-time result preview that updates as the user types the query.

the `Result` operator represents the evolution of this concept into a more declarative, powerful, and manageable framework. The mechanisms for creating, composing, and intelligently managing these cached `Results` are precisely the "sophisticated local caching strategies" required for the next generation of the Instant SQL feature. Therefore, the research in this thesis on optimizing the composition of `Results` is directly driven by the need to advance this real-world application, providing a fast, fluid, and powerful interactive query experience for data analysts.

2.2 The `Result`: A Declarative Caching Mechanism

To tackle the challenges of latency and data locality, a declarative caching mechanism called `Result` was introduced in prior work [NA23].

Conceptually, a `Result` can be understood as a specialized **client-side, unmaintained**

materialized view that is designed to hold only a partial subset of its complete query result.

These properties make a `Result` an ideal mechanism for accelerating interactive data applications, providing the speed of materialized data without the maintenance overhead or the requirement to transfer the complete dataset.

Users create these objects using an intuitive SQL-like syntax, such as `CREATE RESULT R AS Q`, where `R` is the result name and `Q` is a query. Once a `Result` is defined, the system populates it in the background with minimal impact on other running queries. When a subsequent query accesses the `Result`, the query optimizer transparently chooses one of three execution strategies:

1. **Sufficient Cache:** If the local cache already contains enough data to satisfy the entire query (e.g., a `SELECT * FROM R LIMIT 10` when the cache has at least 10 rows), the query is answered instantly from the local cache.

2. **Pending Cache:** If the cache does not yet contain sufficient data but is expected to in a short time, the query can wait. This is only deemed safe if the query has **guaranteed satisfiability**—meaning it explicitly states the number of tuples it needs (e.g., via a `LIMIT` clause) and this number is below a resource-dependent threshold.

3. **Insufficient Cache:** If the query cannot be guaranteed to be satisfied by the cache (e.g., it has no `LIMIT` or a very large one), the optimizer bypasses the cache. It rewrites the query to use the original `Query(R)` as a subquery, effectively treating the `Result` like a standard view and executing the request against the server.

Note that `Results` are not maintained under updates. Users explicitly accept these semantics by choosing to use the `Result` syntax.

While the `Result` operator is a powerful tool for increasing data locality, its initial design is constrained by a fundamental principle: it must always provide **complete and correct results** for the queries it can accelerate.

This strict completeness guarantee limits the scope of queries that can benefit from the local cache. The optimizer can use the cache to answer an incoming query of the form `SELECT <projection> FROM R LIMIT N` only if the number of requested rows, N, is less than or equal to the number of rows either currently stored in Cache(R) or that are guaranteed to be eventually available. For any other operation—such as performing a `JOIN` between two `Results`, applying aggregations, filters, or other complex transformations—the system must bypass the local cache and recompute the fully resolved query on the server.

This thesis proposes a significant extension to this framework by adding new optimizer rules that allow for the creation of a new `Result`, *R*, from a query *D* that operates on the existing caches of other `Results` ($R_1...R_n$). The key innovation that enables this is the **relaxation of the completeness requirement**.

The new cache, `Cache(R)`, that we compute from these partial inputs does not have to be a complete answer to the full `Query(R)`. It only needs to be a valid **subset** of that full result. For the purpose of generating an interactive preview, this partial result is considered sufficient as long as its number of rows is above a given threshold. This philosophical shift is powerful: it allows us to treat the local `Cache(Ri)` objects as in-memory samples, greatly expanding the amount and complexity of queries that can be executed instantly on the client side.

2.3 The Core Problem: Insufficient Cardinality from Composed Caches

As established in the previous section, the central contribution of this thesis is to extend the `Result` framework by relaxing the strict completeness guarantee. This powerful shift enables the next logical frontier and the primary topic of this work: the ability to **compose** `Results` by executing queries directly on the partial, local caches of existing `Results`.

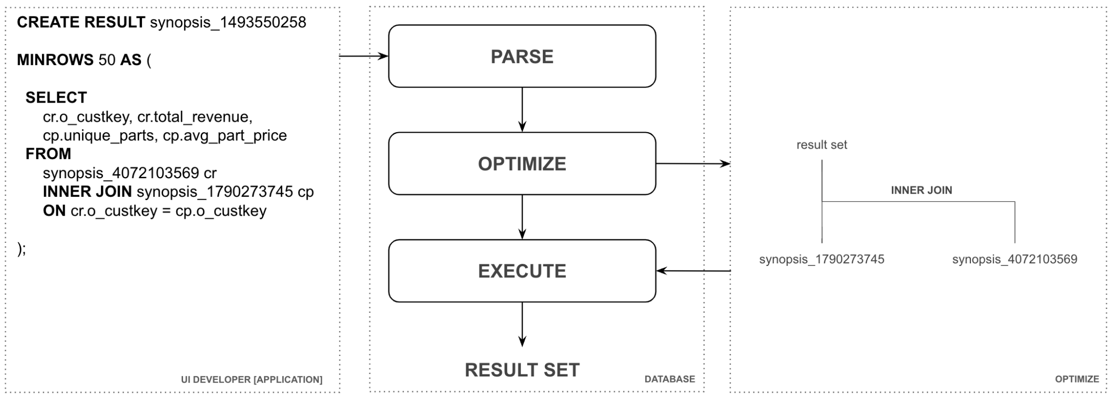
While this approach allows for complex, multi-stage analyses to be performed instantly on the client, it also introduces a new and fundamental challenge: **insufficient result cardinality**. When a query is executed over `Cache` objects, which are themselves partial samples of the full data, the resulting dataset can be dramatically smaller than the inputs. A `JOIN` between two small samples, for example, may yield very few rows or even an empty set.

To make this approximate query processing practical for interactive use cases like Instant SQL, we must be able to specify a minimum acceptable result size for a preview. For this, we propose a `MINROWS Y` clause.

The recovery process follows a strategy that is reminiscent of *Adaptive Query Processing*. Rather than committing to a single fixed plan, the system dynamically reacts to feedback observed during execution.

Concretely, the process begins with the query plan that uses only cached Results (Figure 2.3). This corresponds to the most optimistic case: if the cache contents are sufficient, the query preview can be produced instantly without touching the base tables. However, if during execution we observe that the number of tuples produced falls below the required `MINROWS`, the system aborts and returns to the planning phase.

At this point, the query plan is revised by selectively replacing one or more cached Results with their full underlying query definitions. The revised plan is then re-executed in the hope of producing enough tuples to satisfy the `MINROWS` constraint. Two



```
CREATE RESULT synopsis_1493550258

MINROWS 50 AS (

  SELECT
    cr.o_custkey, cr.total_revenue,
    cp.unique_parts, cp.avg_part_price
  FROM
    synopsis_4072103569 cr
    INNER JOIN synopsis_1790273745 cp
    ON cr.o_custkey = cp.o_custkey

);
```

Figure 2.3: Initial plan using only cached Results (no base-table access). If the tuples produced meet `MINROWS`, the preview completes without re-planning.

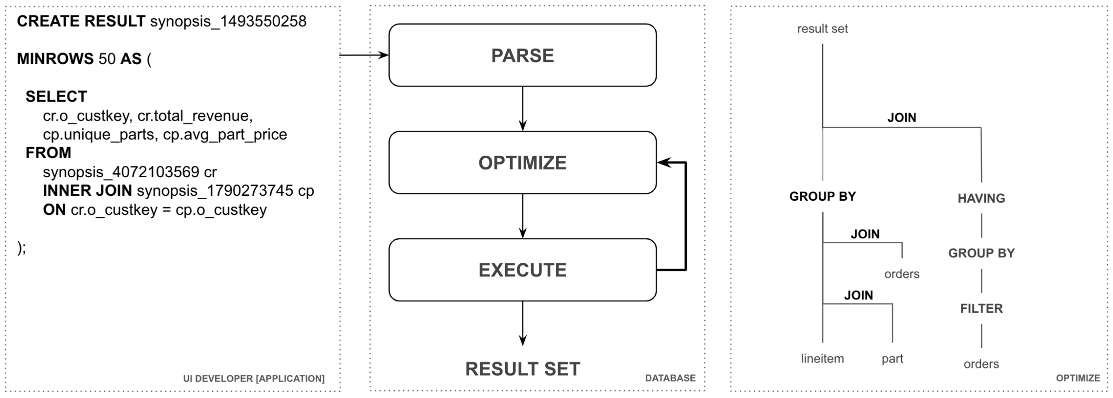
extremes define the space of recovery strategies. At one end, all caches can be bypassed and substituted with their fully resolved queries down to the base tables. This "naive" approach (Figure 2.4) guarantees completeness but incurs the highest cost. At the other end, the system can attempt to re-create only a single cached Result whose expansion is predicted to yield sufficient additional rows. This targeted approach (Figure 2.5) can provide a much faster recovery while still meeting the `MINROWS` requirement.

In this way, the recovery mechanism combines speculative execution with adaptive re-optimization. Like Adaptive Query Processing, it uses runtime feedback to refine the query plan and balance efficiency with the need to maintain meaningful output cardinality. This leads directly to the core research problem of this thesis:

What is the optimal recovery strategy when a `Result` composed from other caches yields a result set with fewer rows than `MINROWS Y`?

2.4 Illustrative Example: The `MINROWS` Failure and Recovery

To illustrate the core problem addressed by this thesis, we will use a query sequence from the `tpch_19.sql` session file used in our experiments. This example demonstrates how a `MINROWS` failure occurs and contrasts the different recovery strategies the system can employ.



```
CREATE RESULT synopsis_1493550258

MINROWS 50 AS (

 SELECT
    cr.o_custkey, cr.total_revenue,
    cp.unique_parts, cp.avg_part_price
 FROM
    synopsis_4072103569 cr
    INNER JOIN synopsis_1790273745 cp
    ON cr.o_custkey = cp.o_custkey

);
```

Figure 2.4: Naive recovery: bypass all caches and substitute fully resolved queries down to base tables. This guarantees completeness but typically incurs the highest cost.

Setting the Scene: An Analyst's Workflow

In this session, an analyst first creates `recent_lineitems` from the base table `lineitem`. Next, they create `priority_orders` from the base table `orders`. Since the base tables reside on the server, these initial queries are executed remotely, and a subset of each result is cached on the client. The analyst then proceeds to build `lineitems_with_order_details` by joining these cached results entirely on the client side. Then, a series of composed `Results` are created, creating the dependency graph shown in Figure 2.6.

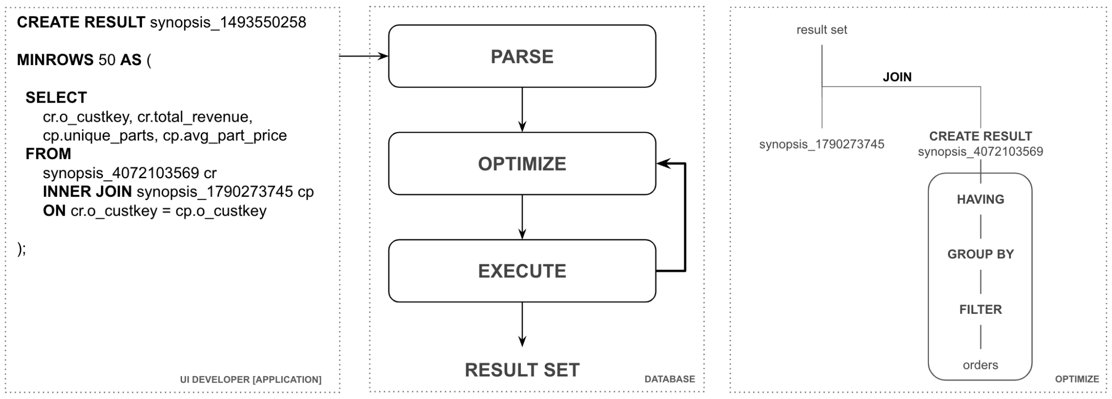
The analyst successfully creates `lineitems_with_order_details`, `final_transactions_cte`, `transactions_for_analysis`, and `filtered_parts`, as each step yields more than the session's requirement of `MINROWS 50`.

The Failure Event

The problem arises when the final `Result`, `transaction_part_details`, is created. This query, which joins `filtered_parts` and `transactions_for_analysis`, produces only **13 rows**. This falls far below the minimal threshold of 50, triggering a `MINROWS` failure and requiring the system to automatically find a recovery path.

Exploring Recovery Strategies

Faced with this failure, the system has several options:



```
CREATE RESULT synopsis_1493550258

MINROWS 50 AS (

  SELECT
     cr.o_custkey, cr.total_revenue,
     cp.unique_parts, cp.avg_part_price
  FROM
     synopsis_4072103569 cr
     INNER JOIN synopsis_1790273745 cp
     ON cr.o_custkey = cp.o_custkey

);
```

Figure 2.5: Targeted recovery: re-create only the selected cached Result predicted to add enough rows quickly, then re-execute the revised plan. This aims to satisfy `MINROWS` at lower cost than the naive approach.

1. The Baseline Approach: Full Replacement The simplest recovery method is to bypass all local caches and re-write the query for `transaction_part_details` into its fully resolved form. This involves expanding every `Result` in its dependency graph into its underlying SQL definition as a subquery, as illustrated in Figure 2.7. This single, monolithic query is then executed on the server.

- **Final Rows:** 50,000 (the maximum cache size)
- **Response Time:** 0.85 seconds

This approach is successful in meeting the threshold but is slow, representing the "brute-force" performance baseline.

2. The Targeted Approach: Re-creating Dependencies A more intelligent approach is to re-create one of the viable candidates (the transparent boxes) to increase its local row count, and then re-execute the final query locally. Viable candidates refer to cached results that are not fully populated and are expected to return additional rows upon re-creation. The outcomes of targeting each candidate are:

- **Candidate `transactions_for_analysis`:** Re-creating this `Result` (visualized in Figure 2.8) and then re-running the final query locally yields **655 rows** with a response time of **0.47 seconds**.
- **Candidate `final_transactions_cte`:** This was the optimal choice. Re-creating it (Figure 2.9) yields **641 rows** with a response time of only **0.45 seconds**.

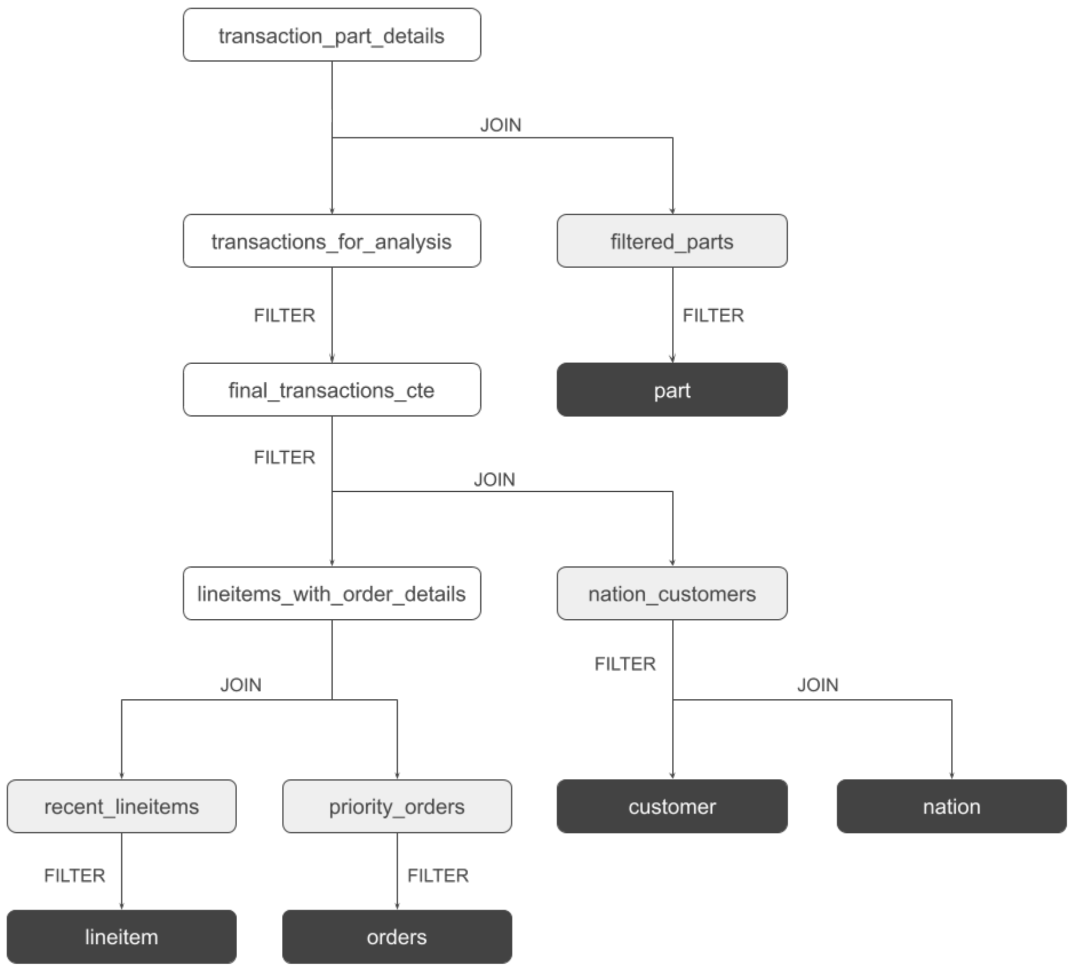


Figure 2.6: The initial dependency graph at the moment of the `MINROWS` failure. Dark boxes are base tables, grey boxes are non-candidate `Results`, and transparent boxes are viable candidates for re-creation.

- **Candidate** `lineitems_with_order_details`**:** This represents a poor choice, illustrated in Figure 2.10. Despite its lower re-creation cost, increasing its size did **not** increase the final number of rows.

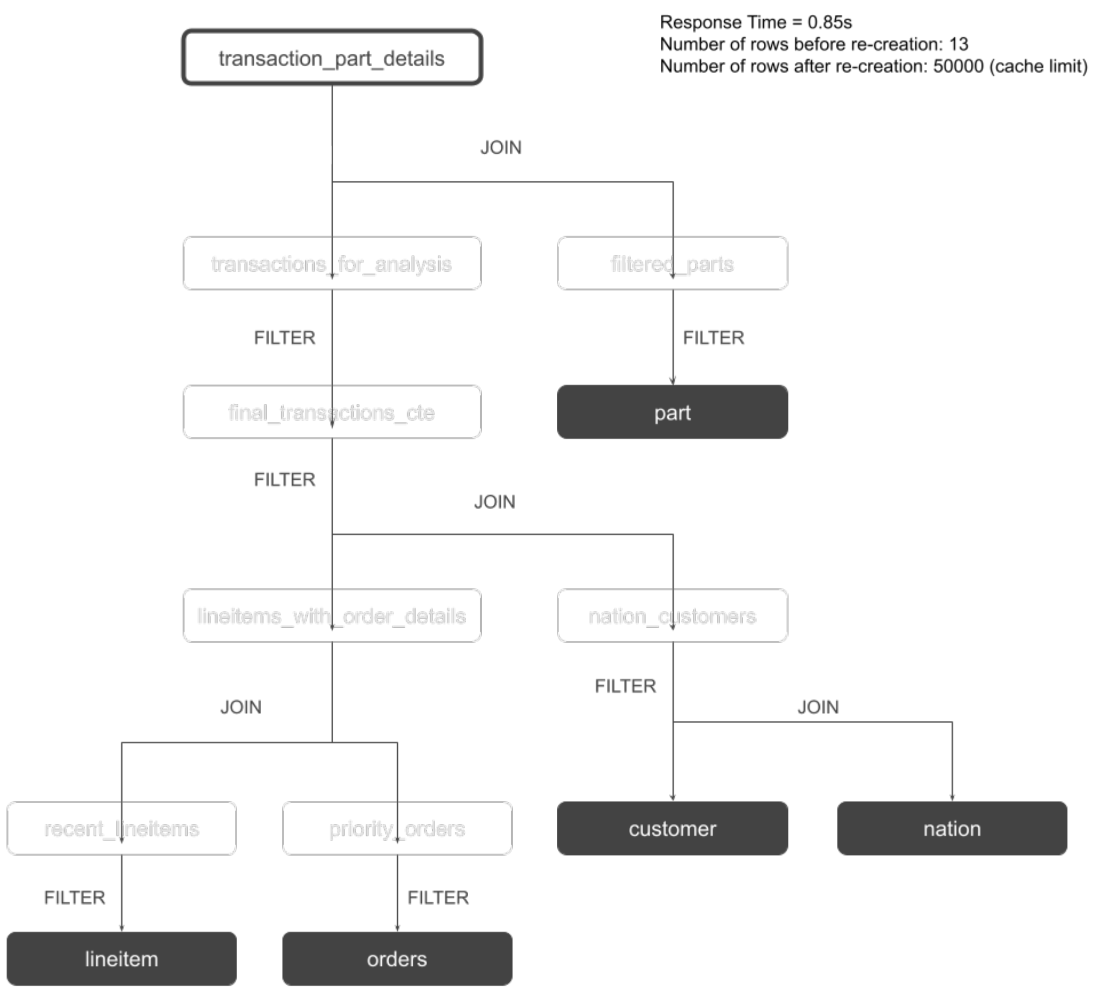


Figure 2.7: Diagram illustrating the Full Replacement strategy, where the final query resolves all dependencies back to the base tables.

The Central Research Question

This scenario crystallizes the core problem addressed by this thesis. The system must choose between a slow but guaranteed baseline and several cheaper but uncertain targeted re-creations. One candidate (`final_transactions_cte`) is optimal, while another (`lineitems_with_order_details`) is ineffective. This leads to the central research question:

How can the system automatically and intelligently predict the outcome of re-creating each

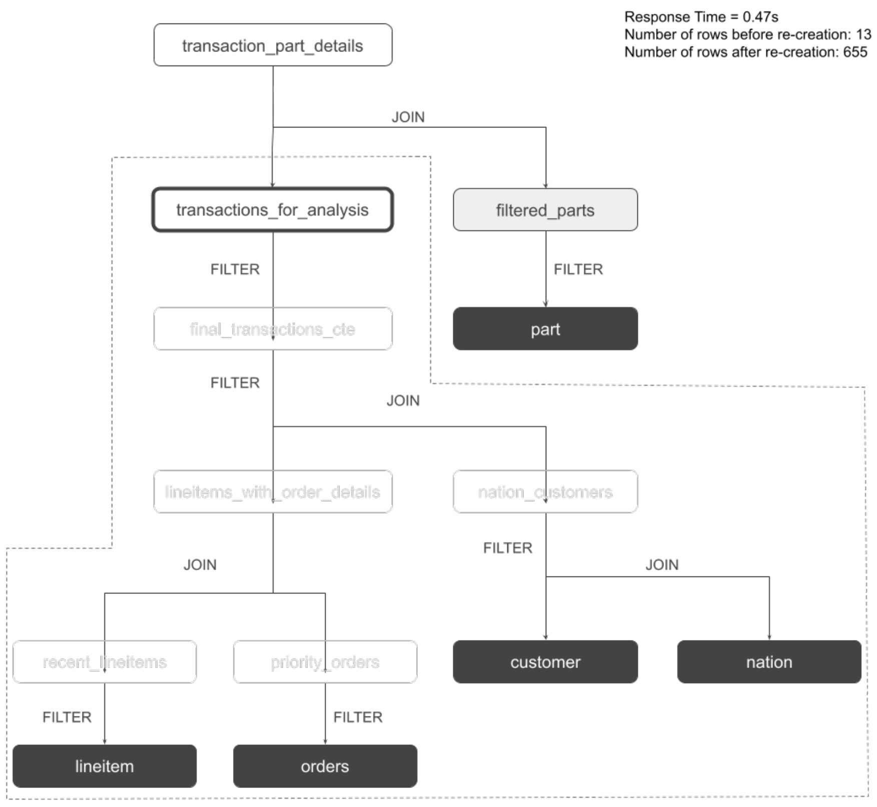


Figure 2.8: Re-creation of `transactions_for_analysis`.

candidate to select the optimal one—the one that satisfies the `MINROWS` constraint at the lowest possible cost—without resorting to the slow baseline or making an ineffective choice?

2.5 Competing Strategies

2.5.1 Strategy A: Full Replacement

The **Full Replacement** strategy serves as the ultimate fallback mechanism, guaranteeing correctness at the expense of performance.

The mechanism involves a process of **full query resolution**. The system takes the original query written against the `Result` objects and recursively expands each `Result` identifier (R_i) into its full underlying `Query(`R_i`)`. This expansion continues down the dependency graph until the final query references only the base tables on the server.

This approach is characterized by several significant drawbacks:

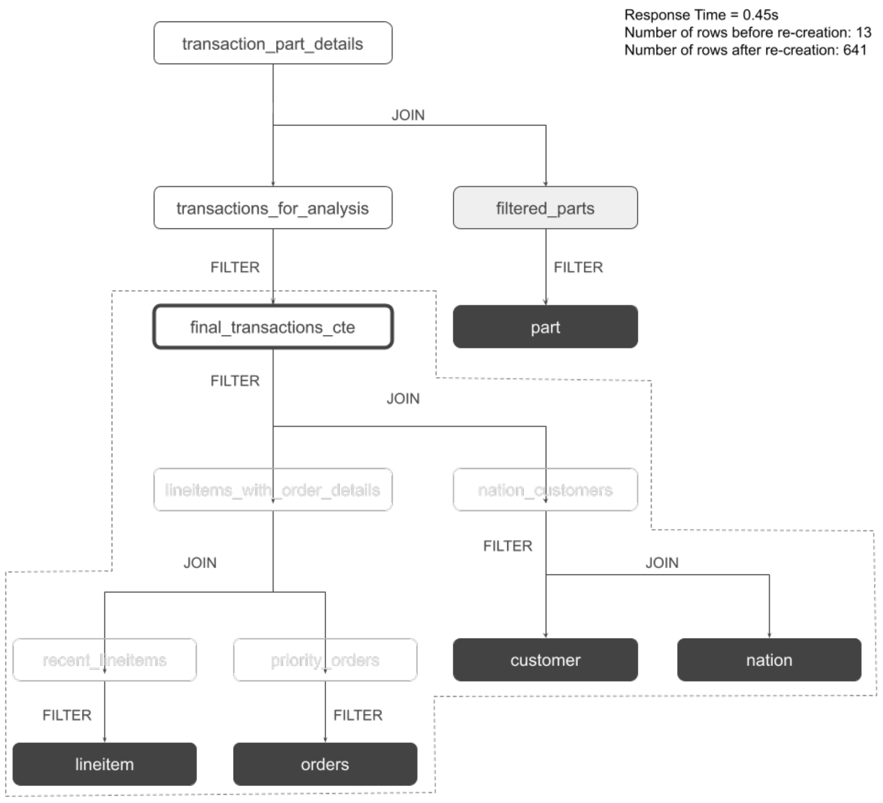


Figure 2.9: Re-creation of the optimal candidate, `final_transactions_cte`.

- **Maximum Cost:** It incurs the highest possible computational cost by running the most complex version of the query on the server, performing no local computation.
- **High Latency:** It maximizes user-perceived latency, as it involves a full network roundtrip for a potentially long-running query.
- **Negation of Architecture:** It completely negates the core benefits of the hybrid architecture, rendering the local caches and compute capabilities useless for this query.

Therefore, this strategy acts as a crucial but undesirable **performance baseline**. It represents the "brute-force" approach, providing a benchmark against which the efficiency and intelligence of the "Targeted Re-creation" strategy can be measured.

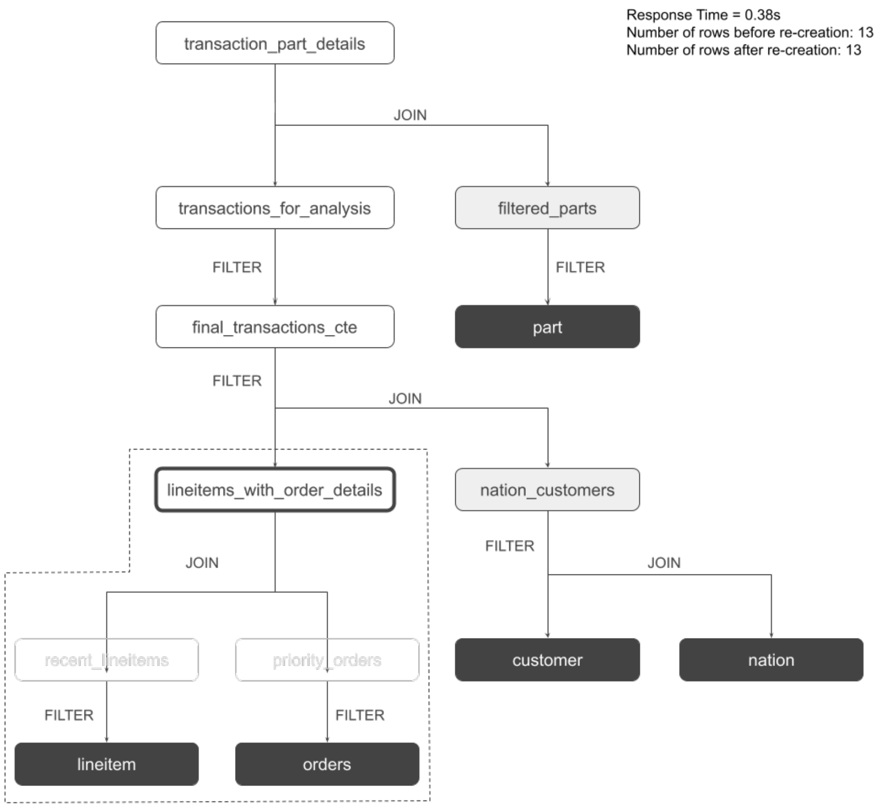


Figure 2.10: Re-creation of the ineffective candidate, `lineitems_with_order_details`.

2.5.2 Strategy B: The Targeted Re-creation Approach

The targeted re-creation strategy is the intelligent alternative to the brute-force baseline. Instead of re-computing the entire query, the system attempts a more surgical fix by re-creating a single dependency (a candidate `Result`) to fix the `MINROWS` failure.

Using our example where `transaction_part_details` failed, the system first identifies all viable candidates (the transparent boxes in Figure 2.6). For each one, it must predict two key outcomes: the final number of rows it would produce and the cost (response time) of the re-creation. In this scenario, the system's analysis would yield the following options:

- **Option 1 (`transactions_for_analysis`):** Predicted to yield **655 rows** at a cost of **0.47s**.
- **Option 2 (`final_transactions_cte`):** Predicted to yield **641 rows** at a cost of **0.45s**.

- **Option 3 (`lineitems_with_order_details`):** Predicted to yield no increase in rows, remaining at 13.

The system then applies its **hierarchical decision-making logic**:

1. **Filter for Success:** It first discards any option not predicted to meet the `MINROWS 50` threshold. This immediately disqualifies `lineitems_with_order_details`.
2. **Select Lowest Cost:** It then compares the costs of the remaining successful candidates. Since 0.45s is cheaper than 0.47s, the system selects `final_transactions_cte` as the optimal choice.

Once the choice is made, the server re-computes `final_transactions_cte` and returns the larger cache to the client. The client then performs a **local cascading update**, re-running the queries for `transactions_for_analysis` and the final `transaction_part_details`. This successfully produces 641 rows, satisfying the constraint at nearly half the cost of the baseline approach.

This example demonstrates the power of the targeted approach. However, its success is entirely dependent on the system's ability to generate accurate predictions. Not only must it estimate the final cardinality, but it must also correctly **estimate the cost** of each potential re-creation to make the optimal choice. The following chapter will explore the various strategies developed to produce these critical estimates.

2.6 Cost Estimation for Targeted Re-creation

The hierarchical selection model described previously relies on a crucial input: the estimated cost of re-creating each candidate `Result`. Accurately predicting the execution time of a query is a notoriously difficult problem, even for sophisticated database optimizers with access to full table statistics. In our context, this challenge is amplified, as the decision must be made on the client side with access only to partial, cached data.

However, for our selection heuristic, obtaining the exact execution cost in seconds is not strictly necessary. The primary requirement is the ability to reliably **sort** the candidates by their expected cost to identify the least expensive option. This insight allows us to develop a simple yet effective heuristic based on a `Result`'s position within the dependency graph.

2.6.1 Depth as a Proxy for Cost

We propose a heuristic that uses a `Result`'s **depth as a proxy for its re-creation cost**. We define depth as the longest path from a `Result` back to a base table in the dependency graph. A Result created directly from a base table is at the deepest level, while the final Result in a long chain is at the shallowest.

The logic is that the cost of re-creating a Result is proportional to the complexity of its fully resolved query. A deeper Result (closer to a base table) has a simpler resolved query with fewer transformations. A shallower Result (higher up the graph) has a more complex resolved query that must incorporate all the transformations of its ancestors. Therefore, we assume:

$$\textit{Deeper } \texttt{Result} \Rightarrow \textit{Simpler Resolved Query} \Rightarrow \textit{Lower Re-creation Cost}$$

It is important to acknowledge that this heuristic is not infallible. A sophisticated server-side optimizer could potentially take a complex query for a shallow Result and push down a highly selective filter, making it execute faster than the query for a deeper Result. However, as our experimental results demonstrate, this heuristic proves to be an effective and reliable proxy for the majority of analytical workloads.

2.6.2 Comparing Across Lineages and The Final Selection Model

A critical constraint of this heuristic is that it can only be used to compare the costs of Results that belong to the **same lineage**—that is, Results that trace back to the same set of base tables. We cannot compare the depth of Results from different lineages, as their query complexities are not comparable.

For example, in Figure 2.6, we can assume that re-creating final_transactions_cte is more expensive than re-creating its ancestor, lineitems_with_order_details. However, we cannot make any cost assumption between lineitems_with_order_details and nation_customers, as they belong to different lineages originating from different base tables.

This constraint leads to a refinement of our final selection model:

1. **Filter for Viability:** First, identify all candidate Results that are predicted to satisfy the MINROWS constraint (Y), forming the set of viable candidates, $\mathcal{V}$.

$$\mathcal{V} = \{R_i \in \text{Candidates} \mid \text{PredictedCard}(R \mid \text{re}-\text{create}R_i) \geq Y\}$$

2. **Group by Lineage:** Group these viable candidates by their lineage set, $\mathcal{L}$.

3. **Find Lineage Champions:** For each lineage $L \in \mathcal{L}$, select the cheapest candidate using our depth-as-cost heuristic (i.e., the one with the greatest depth). This produces a set of "finalist" candidates, $\mathcal{C}^*$.

$$\mathcal{C}^* = \{\arg\max_{R_i \in \mathcal{V}_L} \text{Depth}(R_i) \mid L \in \mathcal{L}\}$$

where $\mathcal{V}_L$ is the subset of viable candidates from lineage L.

4. **Select for Utility:** From this set of finalists, the system selects the final winner, R^*, that is predicted to produce the **maximum number of rows**.

$$R^* = \arg\max_{R_j \in \mathcal{C}^*} \text{PredictedCard}(R \mid \text{re}-\text{create}R_j)$$

This final step optimizes for **utility**. By choosing the candidate that most significantly enriches the local cache, we increase the likelihood that future queries in the same session can be answered without triggering another `MINROWS` failure, thereby reducing the total number of re-creations over the entire session. Also, this approach minimizes the risk of a "false positive," a scenario where our chosen candidate, upon re-creation, still fails to meet the required threshold. A candidate predicted to produce a large number of rows provides a greater buffer against estimation errors, making it a more reliable choice for the immediate recovery.

If our estimates claim that no candidate can lead to the minimal number of rows upon re-creation, we immediately fall back to the baseline strategy. Similarly, if our attempt at re-creating a chosen candidate also fails to produce enough rows, we re-execute the query again, this time using the baseline strategy.

This means that choosing the wrong candidate leads to significant **overhead**, as more than one server-side execution is required. However, it is important to note that selecting the wrong cached result is not a total waste. The failed attempt still enriches the cache of the chosen dependency and its parents, increasing the **utility** of these intermediate results for potential re-use in future queries.

Still, selecting the *right* candidate on the first try is crucial for preserving the user's perception of a fast, interactive system.

The optimization strategy is guided by a clear hierarchy of objectives, designed to provide immediate responsiveness while also promoting long-term session efficiency.

Perceived Cost of a Single Re-creation Event

The perceived cost of a single recovery event, C_{event}, is the total time a user must wait. It is the sum of network latency (T_{network}), server execution time (T_{server}), and the client-side computational overhead introduced by each strategy's decision-making process (T_{client}).

$$C_{event} = T_{\text{network}} + T_{\text{server}} + T_{\text{client}}$$

Total Session Cost

The total cost for an entire session, $C_{session}$, is the sum of all N individual recovery events.

$$C_{session} = \sum_{j=1}^{N} C_{event,j}$$

Optimization Goals

Our model addresses two goals in hierarchical order:

1. Primary Goal: Minimize Immediate Cost The primary objective is to minimize the cost of the current recovery event, C_{event}. This directly addresses the immediate user-perceived latency and is crucial for maintaining a responsive experience. Our selection model targets this by using depth as a proxy to identify the set of lowest-cost candidates from each lineage.

2. Secondary Goal: Minimize Total Session Cost The secondary objective is to minimize the cumulative cost, $C_{session}$, by reducing the total number of re-creations, N, throughout the session. Our model addresses this by applying the **utility heuristic**: from the set of cheapest candidates, it selects the one predicted to produce the most rows. This choice is intended to best enrich the local caches, reducing the likelihood of future `MINROWS` failures.

3 Targeted Re-creation Strategies

The success of the "Targeted Re-creation" strategy, as defined in the previous chapter, hinges entirely on the ability to accurately predict the outcome of a potential re-creation. When faced with an insufficient result, the system must decide which single dependency to re-create by executing its full query on the server. This decision is guided by a heuristic that requires a reliable estimate of the final result's cardinality for each potential candidate.

Specifically, the core predictive challenge is to answer the question: "If we invest the cost to re-create `Cache(R_i)`, how many rows will the final `Result`, R, have after the cascading client-side update?" An accurate answer allows the system to make an informed, cost-effective decision. This chapter provides an overview of three distinct approaches for generating these crucial cardinality estimates, each representing a unique trade-off between speed, accuracy, and complexity.

3.1 Overview of Prediction Strategies

The success of targeted re-creation depends on accurately predicting the outcome of potential re-creations. We identify three distinct approaches for generating cardinality estimates:

- **Strategy 1: Optimizer Cardinality Estimation** - Leverages DuckDB's internal optimizer statistics for fast, but potentially inaccurate estimates.
- **Strategy 2: Local Sampling and Regression** - Uses local cache data with regression analysis to predict cardinality gains.
- **Strategy 3: Data Provenance** - Analyzes the influence of input tuples on the failed computation to identify critical dependencies.

Each strategy represents a unique trade-off between speed, accuracy, and complexity, as detailed in the following sections.

The strategies below are grounded in a formal definition of the state of each cached result.

First, we define three mutually exclusive states for any given `Result`, R:

- **Not full and complete:** The cache `Cache(R)` has not reached its maximum size (`MaxRows`), and all its ancestor `Results` were themselves complete (or were created from base tables). A `Result` in this state represents a complete query result.
- **Full and incomplete:** The cache `Cache(R)` is at its maximum size, so it cannot grow further, regardless of the state of its ancestors.
- **Not full and incomplete:** The cache `Cache(R)` has space to grow, but it was created from at least one ancestor that was incomplete. This means `Cache(R)` is only a partial result of a partial result.

Under this model, the only viable candidates for a re-creation effort are `Results` in the **"Not full and incomplete"** state. A "Not full and complete" result will not change, and a "Full and incomplete" result has no space to store more tuples. Only the "Not full and incomplete" results have both the potential to grow and a reason to do so.

3.2 Optimizer Cardinality Estimation

This approach leverages the internal mechanisms of the DuckDB query optimizer. For each potential re-creation, we can construct a hypothetical query plan and ask the optimizer to *estimate* the final cardinality based on its stored statistics and models. This method is very fast as it processes no actual data, but its accuracy is subject to the inherent limitations of cardinality estimation in complex queries.

3.2.1 Mechanism

The optimizer-based approach works by constructing a hypothetical query that represents the scenario where a specific `Result` R_i has been re-created with its full dataset. This query is then submitted to DuckDB's query optimizer, which applies its internal cardinality estimation models to predict the final result size.

The key insight is that we can simulate the effect of re-creating R_i by replacing its cache reference with the original query definition, limited by the cache size. For example, if R_1 was defined as `SELECT * FROM users WHERE country = 'DE'`, re-creating it would be equivalent to running the final query with this subquery in place of the cached result, constrained by the maximum tuple limit permitted for cached results.

Optimizer-Based Recovery (Cardinality + Cost Selection)

Idea. For each candidate cached *Result* R_i, we build a hypothetical query by *inlining* R_i's original definition (bounded by the cache tuple limit) into the final query. The optimizer is then used to estimate the cardinality of the resulting plan, while the cost of re-creating R_i is obtained from the cost function defined previously. Among candidates whose estimated cardinality is at least `MINROWS`, we choose the one with the lowest cost.

Inputs.

- `Q_final`: the user's final query (may reference cached Results $\{R_i\}$).
- `Candidates` $= \{R_i\}$: upstream Results eligible for re-creation.
- `Def`[R_i]: original SQL definition of each R_i (pre-caching).
- `CACHE_LIMIT`: max tuples allowed when re-creating a Result.
- `MINROWS`: minimum acceptable final cardinality after recovery.
- `COST`(R_i): cost function returning the cost of re-creating R_i.

Helper functions.

- `INLINE_WITH_BOUND(Q, Ri)`: replace each reference to R_i in `Q` with

  ```
  (SELECT * FROM (Def[Ri]) AS _ri LIMIT CACHE_LIMIT)
  ```

- `ESTIMATE_CARD(Q)` $\rightarrow$ `est_rows`: obtain the optimizer's estimated output cardinality for `Q` (e.g., via `EXPLAIN` / planner API).

Pseudo-code.

```
ALGORITHM CHOOSE_RECREATION_BY_CARD_AND_COST(Q_final, Candidates):

  best_candidate  := NULL
  best_cost       := +INF
  best_est_rows   := 0

  for each Ri in Candidates:
      Q_hat := INLINE_WITH_BOUND(Q_final, Ri)
      est_rows := ESTIMATE_CARD(Q_hat)
      est_cost := COST(Ri)

      -- accept only candidates predicted to meet the minimum cardinality
      if est_rows >= MINROWS then
```

```
        -- among feasible ones, prefer the lowest cost
        if est_cost < best_cost then
            best_candidate := Ri
            best_cost      := est_cost
            best_est_rows  := est_rows
        end if
    end if
end for

if best_candidate != NULL then
    return (best_candidate, best_est_rows, best_cost)
else
    -- no candidate predicted to meet MINROWS
    return (NULL, 0, +INF)
end if
```

Example inline transformation. If R_1 was `SELECT * FROM users WHERE country='DE'`, any reference to `R1` in `Q_final` becomes:

```
(SELECT * FROM (SELECT * FROM users WHERE country='DE') AS _r1 LIMIT CACHE_LIMIT)
```

Decision rule. Select the candidate R_i whose hypothetical plan is estimated to produce at least `MINROWS` rows and has the lowest cost according to the function `COST(`R_i`)`. In case of a cost tie, prefer the candidate with higher `est_rows`.

3.2.2 Advantages

- **Speed:** No data transfer or computation required; estimates are generated purely from statistics.
- **Integration:** Leverages existing, well-tested cardinality estimation infrastructure.
- **Consistency:** Uses the same estimation models that guide query optimization decisions.

3.2.3 Limitations

- **Accuracy:** Inherits the limitations of cardinality estimation in complex queries, especially with joins and aggregations.

- **Statistics Dependency:** Quality of estimates depends on the freshness and accuracy of table statistics.
- **Model Assumptions:** Relies on statistical models that may not capture complex data correlations.
- **Client-Side Statistics Requirement:** For this approach to be feasible, table statistics must be cached on the client side. Otherwise, requesting statistics from the server would incur network latency, making the optimizer-based approach impractical for interactive applications.

3.3 Estimation via Local Sampling and Regression

This strategy moves away from statistical estimates and instead proposes an empirical method to predict cardinality. It uses the data already present in the local caches to build a predictive model for each dependency's influence on the final result.

3.3.1 Estimation Methodology

When an initial query fails to meet its `MINROWS` target, the system identifies all direct children in the "Not full and incomplete" state. For each such candidate child, R_i, it performs the following procedure to determine its influence:

1. **Iterative Sub-sampling:** The system takes multiple Bernoulli samples of the existing local cache, $\texttt{Cache(R}_\texttt{i}\texttt{)}$, at varying percentages (e.g., 25%, 50%, and 75%).
2. **Re-computation:** For each sample, it re-executes the final query D, using the sampled $\texttt{Cache(R}_\texttt{i}\texttt{)}$ and 100% of all other dependency caches. The number of rows produced in the final result is recorded for each run.
3. **Linear Regression:** Using the data points from these runs, plus the point from the original failed attempt (100% of $\texttt{Cache(R}_\texttt{i}\texttt{)}$), the system performs a linear regression. The x-axis represents the number of input tuples from $\texttt{Cache(R}_\texttt{i}\texttt{)}$, and the y-axis represents the number of output tuples in the final result, R.
4. **Influence as Slope:** The **slope** of this regression line is a powerful indicator. It represents the "yield" or "influence factor" of R_i, quantifying how many output rows are produced in R for each single input row from $\texttt{Cache(R}_\texttt{i}\texttt{)}$.
5. **Extrapolation:** The system then calculates the number of additional rows that could be stored in R_i ($\texttt{Space\_left = MaxRows(R}_\texttt{i}\texttt{) - |Cache(R}_\texttt{i}\texttt{)|}$). The total

predicted gain in final cardinality is estimated as `Predicted_gain = Space_left * slope`.

3.3.2 Recursive Application and Selection

This estimation process is applied recursively. The system expands the children in the "Not full and incomplete" state and performs the same sampling and regression analysis on their children. This allows it to evaluate the downstream impact of re-creating a "grandchild" `Result`.

Once this recursive analysis is complete, the system possesses a set of potential re-creation actions and their predicted cardinality gains. It can then apply the hierarchical selection process defined previously: first, it filters this set down to only those actions predicted to satisfy the `MINROWS` constraint, and then it applies the cost-based selection logic to make the final, optimal choice.

Dynamic Sampling Recovery (Sub-Sampling + Regression + Cost Selection)

Idea. When a new cached *Result* is created, we also sample every cached result in its dependency graph (both direct and transitive sources). To decide which upstream R_i to re-create, we sub-sample `Cache(Ri)` at several rates, recompute the final query for each sub-sample (keeping other caches at 100%), fit a simple linear regression of output vs. input, and use the slope to estimate the predicted gain in final cardinality if `Cache(Ri)` were expanded up to capacity. Among candidates predicted to reach at least `MINROWS`, we choose the one with the lowest `COST(Ri)`.

Inputs.

- `Q_final`: the user's final query.
- `Candidates` $= \{R_i\}$: upstream Results eligible for re-creation.
- `Cache(Ri)`: current tuples cached for R_i.
- `MaxRows(Ri)`: capacity limit for R_i.
- `MINROWS`: minimum acceptable final cardinality after recovery.
- `SAMPLE_RATES`: e.g., {`0.25`, `0.50`, `0.75`}.
- `COST(Ri)`: cost function returning the cost of re-creating R_i.

Helper functions.

- `BERNOULLI_SUBSAMPLE(Cache(Ri), p)`: return a Bernoulli p-sample of `Cache(Ri)`.
- `RUN_FINAL(Q_final, Override(Ri -> Sample))`: execute `Q_final` with `Cache(Ri)` replaced by `Sample`; return final output row count.
- `LINEAR_REGRESSION(Points)` $\rightarrow$ `slope`: fit $y \approx \alpha + \beta x$ to the points (input rows vs. output rows) and return β.

Pseudo-code.

```
ALGORITHM CHOOSE_RECREATION_BY_SAMPLING(Q_final, Candidates):

  PredictedGain := {}    -- map: Ri -> predicted additional rows
  Feasible      := {}    -- map: Ri -> boolean

  for each Ri in Candidates:

      Points := []       -- (x = |sample of Cache(Ri)|, y = final rows)

      -- 1) Iterative sub-sampling runs
      for each p in SAMPLE_RATES:
          SampleRi := BERNOULLI_SUBSAMPLE(Cache(Ri), p)
          x := |SampleRi|
          y := RUN_FINAL(Q_final, Override(Ri -> SampleRi))
          append (x, y) to Points
      end for

      -- 2) Include the 100% point (failed attempt or recorded)
      x0 := |Cache(Ri)|
      y0 := RUN_FINAL(Q_final, Override(Ri -> Cache(Ri)))
      append (x0, y0) to Points

      -- 3) Linear regression: slope = influence of Ri
      slope := LINEAR_REGRESSION(Points)

      -- 4) Extrapolation to capacity
      Space_left := MaxRows(Ri) - |Cache(Ri)|
      if Space_left < 0 then Space_left := 0 end if
```

```
    PredictedGain[Ri] := Space_left * slope

    -- 5) Feasibility: projected final >= MINROWS ?
    CurrentFinal := y0
    Feasible[Ri] := (CurrentFinal + PredictedGain[Ri] >= MINROWS)

end for

-- 6) Choose lowest-cost feasible candidate
best_candidate := NULL
best_cost      := +INF
best_gain      := 0

for each Ri in Candidates:
    if Feasible[Ri] == TRUE then
        est_cost := COST(Ri)
        if est_cost < best_cost then
            best_candidate := Ri
            best_cost      := est_cost
            best_gain      := PredictedGain[Ri]
        end if
    end if
end for

if best_candidate != NULL then
    return (best_candidate, best_gain, best_cost)
else
    return (NULL, 0, +INF)
end if
```

Interpretation.

- **Slope as influence.** The slope estimates yield (additional final rows per additional input row from `Cache(Ri)`).
- **Extrapolation.** `PredictedGain[Ri]` = `Space_left` × `slope` estimates the extra final rows if `Cache(Ri)` were expanded to capacity.
- **Decision.** Among candidates predicted to reach `MINROWS`, pick the one with the lowest `COST(Ri)`.

3.3.3 Advantages

- **High Accuracy:** Predictions are derived empirically from the actual data distributions and correlations present in the local caches.
- **No Network Overhead:** Requires no additional network requests to the server.
- **Data-Driven:** Uses actual cache data rather than statistical models or external samples.

3.3.4 Limitations

- **Computational Intensity:** Requires re-executing the final query multiple times for each candidate in the dependency graph, which could be slow if the query is complex or the local caches are large.
- **Local Cache Dependency:** Effectiveness depends on having sufficient data in local caches to perform meaningful regression analysis.
- **Linear Assumption:** Relies on linear regression, which may not capture non-linear relationships in complex queries.

3.4 Estimation via Tuple-Level Data Provenance

This third strategy takes an analytical approach, distinct from both statistical estimation and dynamic sampling. It leverages **data provenance** to create a precise, tuple-level map of how data flows from input caches to the final, insufficient result. By analyzing this lineage, the system can quantify the "influence" of each dependency and predict the impact of a re-creation.

3.4.1 The Provenance Tracking Mechanism

The foundation of this strategy is a built-in provenance tracking mechanism. When any `Result` is created, its cache is augmented with an extra column that stores the provenance for each row.

- **Provenance Sets:** For each tuple, this column contains a **set of references** pointing to the exact source tuples that produced it. Each reference consists of a `{cached result identifier, row identifier}` pair.

- **Propagation:** As new `Results` are created from existing ones, this provenance is propagated. For instance, when a tuple is created by a join between a row from `Cache(R1)` and a row from `Cache(R2)`, its provenance set becomes the **union** of the provenance sets from those two input rows. This ensures that every tuple carries its complete lineage, tracing back to the original source caches.

3.4.2 The Estimation Methodology

When a query fails to meet its `MINROWS` target, the system analyzes the provenance data attached to the small set of resulting tuples. For each candidate dependency R_i (a `Result` in the "Not full and incomplete" state), it computes two key metrics:

1. **Selectivity:** This measures the fraction of R_i's cache that "survives" the query and contributes to the final result. It's calculated as the number of *distinct* tuples from `Cache(R_i)` found in the final result's provenance sets, divided by the total size of `Cache(R_i)`.

$$\text{Selectivity} = \frac{|\text{Distinct contributing tuples from } R_i|}{|\text{Cache}(R_i)|}$$

2. **Expansion Factor:** This measures, on average, how many times each *surviving* tuple from R_i is used to create an output tuple. This is particularly important for joins where a single tuple can have multiple matches.

$$\text{Expansion Factor} = \frac{\text{Total references to tuples from } R_i}{|\text{Distinct contributing tuples from } R_i|}$$

3.4.3 Extrapolation and Selection

With these two factors, the system can predict the number of additional rows that would be generated by filling the remaining space in `Cache(R_i)`:

$$\text{Predicted\_gain} = \text{Space\_left}(R_i) \times \text{Selectivity} \times \text{Expansion Factor}$$

It is important to note that when we expand the terms for Selectivity and Expansion Factor, the "Distinct contributing tuples" term cancels out:

$$\frac{|\text{Distinct contributing tuples from } R_i|}{|\text{Cache}(R_i)|} \times \frac{\text{Total references to tuples from } R_i}{|\text{Distinct contributing tuples from } R_i|}$$

This leaves a much simpler and more efficient formula:

$$\text{Predicted\_gain} = \text{Space\_left}(R_i) \times \frac{\text{Total references to } R_i}{|\text{Cache}(R_i)|}$$

This mathematical simplification has a crucial practical benefit. It means that to calculate the predicted gain, the system only needs to count the total number of times a given `Result` is referenced in the provenance of the final output. It does not need to track the specific row indices to identify distinct tuples. This dramatically reduces the amount of metadata that needs to be stored and propagated with each tuple, significantly lowering the overhead of the provenance mechanism.

This process is applied recursively down the dependency graph to all eligible candidates. Once all predictions are gathered, the system feeds them into the hierarchical selection model described previously to make the final, cost-based decision.

3.4.4 Advantages

- **High Accuracy:** This method can be highly accurate as it is based on the precise, observed behavior of every tuple in the query, under the assumption that the data is uniformly distributed.
- **No Network Overhead:** The estimation phase itself requires no extra network roundtrips or query re-executions.
- **Precise Analysis:** Provides exact information about which input tuples contributed to the output.
- **Query-Agnostic:** Works regardless of the query structure or complexity.

3.4.5 Limitations

- **Computational Overhead:** Storing and propagating provenance sets for every tuple can incur extra memory and CPU cost during the initial query execution, regardless of whether a failure occurs.
- **Memory Requirements:** Requires additional memory to store provenance information for every tuple in the cache.

4 Experiments

This chapter details the empirical evaluation of the three proposed "Targeted Re-creation" strategies. The primary goal of these experiments is to quantitatively measure and compare the efficiency of each approach in a simulated hybrid environment.

To benchmark our proposed heuristics, we also implemented an **Oracle** strategy to establish a theoretical performance limit. For each `MINROWS` failure, the Oracle strategy determines the *exact* number of rows each potential re-creation would yield by executing a `SELECT COUNT(*)` query for every candidate. While this approach is too computationally expensive for a real-world system due to its high overhead, it provides perfect foresight. The results from the Oracle serve as the ideal benchmark for how well any estimation-based strategy could possibly perform.

We also added a **Random** strategy where the cached result to be re-created is selected randomly from the available candidates. This helps validate that our proposed heuristics are truly capturing meaningful signals rather than benefiting from favorable experimental conditions. Also, comparing against random selection demonstrates the practical value of investing computational resources in more sophisticated strategies.

We begin by describing the experimental setup, including the hardware, software, and the synthetic workload used. We then outline the key metrics for evaluation, which focus on the cost and frequency of re-creation events.

4.1 Experimental Setup

To create a controlled and reproducible test environment, we emulated the hybrid client-server architecture and the behavior of the `Result` caching mechanism.

4.1.1 Hardware and Software

All experiments were conducted on a MacBook Pro equipped with an Apple M4 Max processor and 36GB of RAM. The simulation environment was built in Python, leveraging the **DuckDB** library [25b] (version 1.2.0, memory_limit=30GB, threads=8) for all database operations and the **sqlglot** library [21] for programmatic parsing and rewriting of SQL queries according to the logic of each re-creation strategy.

4.1.2 Database and Data

The experimental database was built using the industry-standard **TPC-H benchmark**. The schema and data were generated at **Scale Factor 20**, providing a rich and complex dataset suitable for analytical query workloads.

4.1.3 Hybrid Environment Simulation

To evaluate the different strategies, all experiments were conducted within a Python-based simulation framework[1]. This framework operates by emulating the logic of a real system, programmatically rewriting pure SQL queries according to the rules of each recovery heuristic. The goal of this simulation is to compare the approaches in a controlled setting to identify the most effective strategy for a future native implementation inside the database engine.

The hybrid architecture was simulated as follows:

- **Remote Server:** Any query that accessed the on-disk base tables was considered a "remote" execution. To account for network overhead, a fixed **100ms latency penalty** was added to the measured execution time of every remote query.
- **Client Cache:** Client-side `Cache` objects were simulated using DuckDB's in-memory temporary tables.

4.1.4 Workload Generation

The `Result` feature is a recent development, meaning no real-world user session traces are available for testing. Furthermore, due to the privacy-sensitive nature of user data, we opted to generate a synthetic workload.

Using Large Language Models (LLMs), we generated 597 queries distributed across 20 distinct session files, each designed to simulate realistic data analysis workflows. These queries were crafted to reflect common analytical patterns, including exploratory data analysis, iterative refinement, and multi-step investigations typical of real-world usage. Of these queries, over 98 triggered cache re-creation events, providing sufficient instances to evaluate and compare the performance of each strategy under various scenarios.

[1]The complete source code for the simulation and workload generation is available at: `[https://github.com/marciobbarbosa/results-benchmark]`

4.1.5 Workload Generation Methodology

We adopted a modern approach for workload generation inspired by recent research in LLM-driven benchmarking, such as the SQLStorm methodology [Sch+25]. This approach leverages the fact that Large Language Models (LLMs) have been trained on vast public corpora of SQL code, giving them an implicit understanding of common analytical patterns. By carefully prompting an LLM, we can generate complex, diverse, and realistic query sessions that are well-suited for testing our system.

Interactive Session Generation Framework

We developed a Python-based generation framework that interacts with the Gemini 2.5 Flash API to create entire analytical sessions. This framework goes beyond simple query generation by implementing an interactive validation and correction loop:

1. **Prompt and Generation:** The script sends a carefully engineered prompt to the LLM to generate the first query in a session.
2. **Local Validation:** Upon receiving a query, the script immediately executes it using a local DuckDB instance. This step serves two purposes:
   - **Syntax and Semantic Check:** It validates that the query is executable and syntactically correct.
   - **Non-Empty Result Check:** It verifies that the query produces a non-empty result. An interactive session where every query returns an empty set is not representative of a real user's workflow.
3. **Automated Feedback Loop:** If the query fails validation (syntax error or zero rows), the framework automatically sends the error or a request to "relax the conditions" back to the LLM, asking for a corrected version.
4. **Incremental Session Building:** Once a valid, non-empty query is produced, it is saved as a step in the session. The framework then requests the *next* incremental query for the session. This process is repeated until a complete, multi-step session file is created.

Prompt Engineering and `Result` Emulation

The quality of LLM-generated output is highly dependent on careful prompt engineering. The master prompt, detailed below, was designed to provide the model with a clear persona, objective, and context.

```
Generate the first SQL query in an interactive session simulating a data analyst
working with a TPC-H schema in DuckDB. The session builds queries
incrementally using a sequence of materialized Common Table Expressions (CTEs).

The full TPC-H schema is provided below, including:
- Base table definitions
- Row counts per table
- Value ranges for selected columns

CTE Construction Rules:
- A CTE must select exclusively from either base tables or previously
  defined CTEs, never both.
- Each CTE must be immediately followed by SELECT * FROM <cte_name>;

Session Objective:
Progressively narrow down the dataset toward a focused analytical result. Each
query should incrementally refine the data from the previous step.

Query Guidelines:
- Use typical data analyst patterns.
- Avoid correlated subqueries.
- Output only valid SQL code

Example:

WITH cte1 AS MATERIALIZED (
    SELECT <columns>
    FROM <base_table>
    LIMIT 50000
)
SELECT * FROM cte1;

WITH cte2 AS MATERIALIZED (
    SELECT <columns>
    FROM cte1
    LIMIT 50000
)
SELECT * FROM cte2;

TPC-H schema (rows uniformly and independently distributed):

-- Number of rows: 5
CREATE TABLE region (
    R_REGIONKEY  INTEGER NOT NULL PRIMARY KEY,
    R_NAME       CHAR(25) NOT NULL,
    R_COMMENT    VARCHAR(152)
);

-- Number of rows: 2000000
CREATE TABLE part (
    P_PARTKEY     INTEGER NOT NULL PRIMARY KEY,
    P_NAME        VARCHAR(55) NOT NULL,
    P_MFGR        CHAR(25) NOT NULL,
    P_BRAND       CHAR(10) NOT NULL,
    P_TYPE        VARCHAR(25) NOT NULL,
    P_SIZE        INTEGER NOT NULL, -- min: 1, max: 50
    P_CONTAINER   CHAR(10) NOT NULL,
    P_RETAILPRICE DECIMAL(15,2) NOT NULL, -- min: 900.91, max: 2098.99
    P_COMMENT     VARCHAR(23) NOT NULL
);

-- Number of rows: 25
CREATE TABLE nation (
    N_NATIONKEY  INTEGER NOT NULL PRIMARY KEY,
    N_NAME       CHAR(25) NOT NULL,
    N_REGIONKEY  INTEGER NOT NULL, -- min: 0, max: 4
    N_COMMENT    VARCHAR(152),
    FOREIGN KEY (N_REGIONKEY) REFERENCES region(R_REGIONKEY)
);

-- Number of rows: 100000
CREATE TABLE supplier (
    S_SUPPKEY     INTEGER NOT NULL PRIMARY KEY,
    S_NAME        CHAR(25) NOT NULL,
    S_ADDRESS     VARCHAR(40) NOT NULL,
    S_NATIONKEY   INTEGER NOT NULL, -- min: 0, max: 24
    S_PHONE       CHAR(15) NOT NULL,
    S_ACCTBAL     DECIMAL(15,2) NOT NULL, -- min: -999.92, max: 9999.93
    S_COMMENT     VARCHAR(101) NOT NULL,
    FOREIGN KEY (S_NATIONKEY) REFERENCES nation(N_NATIONKEY)
);

-- Number of rows: 8000000
CREATE TABLE partsupp (
    PS_PARTKEY     INTEGER NOT NULL,
    PS_SUPPKEY     INTEGER NOT NULL,
    PS_AVAILQTY    INTEGER NOT NULL, -- min: 1, max: 9999
    PS_SUPPLYCOST  DECIMAL(15,2)  NOT NULL, -- min: 1.00, max: 1000.00
    PS_COMMENT     VARCHAR(199) NOT NULL,
    PRIMARY KEY(PS_PARTKEY, PS_SUPPKEY),
    FOREIGN KEY (PS_SUPPKEY) REFERENCES supplier(S_SUPPKEY),
    FOREIGN KEY (PS_PARTKEY) REFERENCES part(P_PARTKEY)
);

-- Number of rows: 1500000
CREATE TABLE customer (
    C_CUSTKEY     INTEGER NOT NULL PRIMARY KEY,
    C_NAME        VARCHAR(25) NOT NULL,
    C_ADDRESS     VARCHAR(40) NOT NULL,
    C_NATIONKEY   INTEGER NOT NULL, -- min: 0, max: 24
    C_PHONE       CHAR(15) NOT NULL,
    C_ACCTBAL     DECIMAL(15,2)   NOT NULL, -- min: -999.99, max: 9999.99
    -- segments: AUTOMOBILE, BUILDING, MACHINERY, FURNITURE, or HOUSEHOLD
    C_MKTSEGMENT  CHAR(10) NOT NULL,
    C_COMMENT     VARCHAR(117) NOT NULL,
    FOREIGN KEY (C_NATIONKEY) REFERENCES nation(N_NATIONKEY)
);

-- Number of rows: 15000000
CREATE TABLE orders (
    O_ORDERKEY       INTEGER NOT NULL PRIMARY KEY,
    O_CUSTKEY        INTEGER NOT NULL,
    O_ORDERSTATUS    CHAR(1) NOT NULL, -- status: F, O, or P
    O_TOTALPRICE     DECIMAL(15,2) NOT NULL, -- min: 838.05, max: 558822.56
    O_ORDERDATE      DATE NOT NULL, -- min: 1992-01-01, max: 1998-08-02
    -- priorities: 1-URGENT, 2-HIGH, 3-MEDIUM, 4-NOT SPECIFIED, or 5-LOW
    O_ORDERPRIORITY  CHAR(15) NOT NULL,
    O_CLERK          CHAR(15) NOT NULL,
    O_SHIPPRIORITY   INTEGER NOT NULL, -- min: 0, max: 0
    O_COMMENT        VARCHAR(79) NOT NULL,
    FOREIGN KEY (O_CUSTKEY) REFERENCES customer(C_CUSTKEY)
);

-- Number of rows: 60000000
CREATE TABLE lineitem (
    L_ORDERKEY    INTEGER NOT NULL,
    L_PARTKEY     INTEGER NOT NULL,
    L_SUPPKEY     INTEGER NOT NULL,
    L_LINENUMBER  INTEGER NOT NULL, -- min: 1, max: 7
    L_QUANTITY    DECIMAL(15,2) NOT NULL, -- min: 1.00, max: 50.00
    L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL, -- min: 900.91, max: 104949.50
    L_DISCOUNT    DECIMAL(15,2) NOT NULL, -- min: 0.00, max: 0.10
    L_TAX         DECIMAL(15,2) NOT NULL, -- min: 0.00, max: 0.08
    L_RETURNFLAG  CHAR(1) NOT NULL, -- flags: R, A, or N
    L_LINESTATUS  CHAR(1) NOT NULL, -- status: F or O
    L_SHIPDATE    DATE NOT NULL, -- min: 1992-01-02, max: 1998-12-01
    L_COMMITDATE  DATE NOT NULL, -- min: 1992-01-31, max: 1998-10-31
    L_RECEIPTDATE DATE NOT NULL, -- min: 1992-01-03, max: 1998-12-31
    L_SHIPINSTRUCT CHAR(25) NOT NULL,
    L_SHIPMODE     CHAR(10) NOT NULL, -- modes: REG AIR, MAIL, RAIL, AIR, TRUCK, SHIP, or FOB
    L_COMMENT      VARCHAR(44) NOT NULL,
    PRIMARY KEY(L_ORDERKEY, L_LINENUMBER),
    FOREIGN KEY (L_ORDERKEY) REFERENCES orders(O_ORDERKEY),
    FOREIGN KEY (L_PARTKEY, L_SUPPKEY) REFERENCES partsupp(PS_PARTKEY, PS_SUPPKEY)
);
```

Since the `Result` operator is a novel feature unknown to pre-trained LLMs, we designed a workaround to generate appropriate workloads. The prompt instructs the model to generate a sequence of materialized Common Table Expressions (CTEs), where each subsequent CTE builds upon the previous one. When parsing these generated session files[2], our simulation framework interprets each CTE as a `CREATE RESULT` command. This effectively emulates the dependency graph structure created by a user interactively building `Results` on top of each other.

The prompt was explicitly designed to generate queries that operate **purely on other cached results** (i.e., previous CTEs). This was an intentional choice to isolate the specific class of cache-only compositional queries that are the focus of this thesis's recovery strategies.

Hybrid queries—those that would access both local `Result` caches and remote base tables simultaneously—were not considered for this optimization. Since caches exist only on the client and base tables only on the server, such a query would force the client to upload the (partial) cached data to the server for processing. Given the high overhead of this data transfer and the risk of the query still failing to produce enough rows (as it mixes partial and full data), it is often more practical to execute the fully resolved query on the server directly. Therefore, for the purpose of this thesis, hybrid queries are considered outside the scope of our recovery heuristics and are treated as standard remote queries.

[2]The 20 synthetically generated session files are available at: [https://github.com/marciobbarbosa/results-benchmark/tree/main/sessions]

4.2 Methodology and Metrics

The core of the experiment involves running all 597 queries and, upon encountering a `MINROWS` failure, applying each of the three proposed strategies to select a re-creation candidate. The performance of each strategy is evaluated based on five primary metrics, aggregated across all sessions:

1. **Hit Ratio:** The fraction of re-creation attempts that are successful. Formally,

   $$\mathrm{HR} = \frac{N_{\text{succ}}}{N_{\text{attempts}}}.$$

   A re-creation is *successful* if the strategy re-creates a cached result other than the topmost baseline candidate and, after re-creation, produces $\geq$ `MINROWS` rows.

2. **Total Remote Re-creation Cost (seconds):** This is the sum of the remote execution costs for every re-creation event across all sessions. Each remote execution includes a fixed latency penalty of 100 ms, which models the typical Wide Area Network (WAN) delay observed in cloud-based analytical systems.

3. **Total Local Re-creation Cost (seconds):** This metric captures the computational overhead incurred on the client side when executing each strategy. It includes the time spent on estimation calculations and strategy-specific decision-making processes. This represents the additional processing burden that each approach places on the client system.

4. **Total Re-creation Cost (seconds):** This is the sum of the Total Remote Re-creation Cost and the Local Cost on Client, providing a comprehensive measure of the overall computational expense for each strategy. This metric accounts for both the database-side execution time and the client-side processing overhead, giving a complete picture of the true cost of each cache management approach. It represents the total "wall-clock time" a user would spend waiting for the system to recover from `MINROWS` failures.

5. **Total Number of Re-creations:** This metric counts the total number of times a server-side re-creation was triggered across all sessions. A lower number indicates a more effective strategy at enriching the local cache to prevent future failures.

To ensure a fair and comprehensive comparison, our experiments used two distinct methodologies tailored to the metric being measured.

For metrics evaluating the **Hit Ratio** and **Re-creation Costs** we reset the local cache state after every recovery action. Following a `MINROWS` failure and the application of a given strategy, the local caches were reset to a **common recovered state** before proceeding to the next query in the session. This ensured that all strategies encountered the same `MINROWS` failure while processing identical queries, enabling a controlled, head-to-head comparison of their immediate decision-making intelligence.

In contrast, for the **Total Number of Re-creations** metric, we did **not** reset the cache state to a **common recovered state** . The effects of each recovery—the newly enriched caches—were allowed to persist and influence subsequent queries. The goal of this setup was to measure the long-term, cumulative impact of each strategy and to determine which approach is best at "future-proofing" the session by preventing subsequent failures.

4.3 Results and Analysis

This section presents and analyzes the results of our experiments. We will evaluate each proposed strategy against both the Naive Full Replacement baseline and the theoretical Oracle benchmark across the defined metrics.

The "Dynamic Sampling" strategy was implemented in two distinct variants to explore the fundamental trade-off between predictive accuracy and computational overhead. While both approaches aim to estimate the influence of dependencies by sampling the local cache, they differ significantly in their scope and computational cost.

The first variant, labeled simply **Sampling** in our experiments, represents the most thorough and theoretically pure version of the strategy. When a `MINROWS` failure occurs, this method recursively traverses the dependency graph. For **every potential candidate** (children, grandchildren, etc.) in the "Not full and incomplete" state, it performs the full, expensive estimation process: it runs multiple sub-sample queries and computes a fresh linear regression to derive the slope, or "influence factor," of that candidate relative to the final failed result.

While this approach generates a more accurate, custom-tailored prediction for every possible re-creation action, its primary drawback is the high computational cost. Running numerous query variations for every node in the dependency subgraph can be prohibitively expensive, even on local data, potentially stretching the "free local computation" assumption to an infeasible degree in practice.

To address the performance concerns of the exhaustive approach, we developed a second, more pragmatic variant labeled **Sampling 2**. This strategy drastically reduces the computational workload by limiting the expensive slope calculation.

Instead of analyzing every candidate down the graph from scratch, this method only performs the multi-step sampling and linear regression for the **immediate children** of the failed `Result`. For any deeper dependency (e.g., a "grandchild"), it approximates its influence by re-using previously computed slopes in a hierarchical manner.

The logic is analogous to a chain rule of influence. The impact of a grandchild R_{gc} on the final result R is estimated by multiplying the grandchild's influence on its parent R_c by the child's influence on R:

$$\text{Slope}(R_{gc} \to R) \approx \text{Slope}(R_{gc} \to R_c) \times \text{Slope}(R_c \to R)$$

This makes logical sense: the grandchild's slope represents the impact of a unit increase on its parent, which, when multiplied by the parent's slope, represents the total impact on the final result. This allows the system to generate predictions for deep dependencies quickly, trading the precision of a full re-computation for significant performance gains.

4.3.1 Metric 1: Hit Ratio

This metric isolates the predictive reliability of each strategy. We define a "hit" as an event where a strategy selects a candidate for re-creation that is not the baseline (the top-most `Result`), and that choice successfully produces enough tuples to meet the `MINROWS` requirement. The ratio is the total number of hits divided by the total number of such attempts, measuring how often an "intelligent" choice pays off.

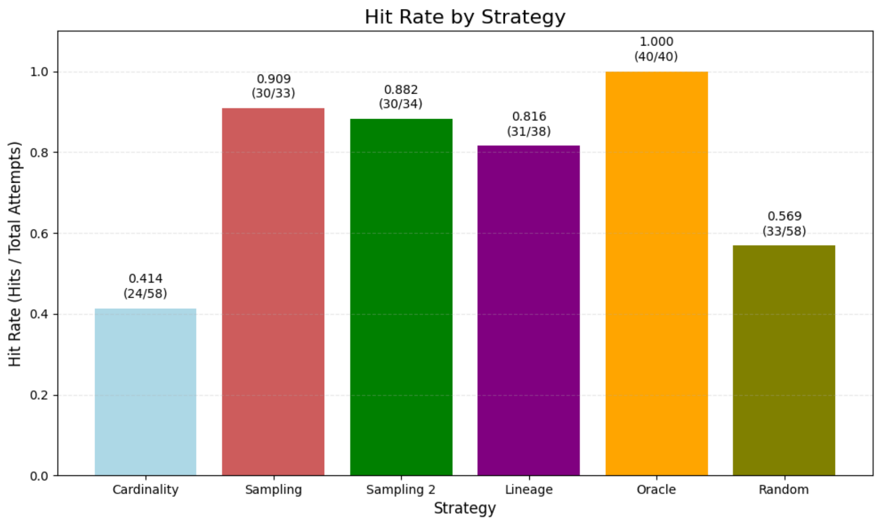


Figure 4.1: Predictive accuracy (Hit Ratio) of each strategy's non-baseline choices.

The results show a clear hierarchy in predictive accuracy:

- **Oracle:** 100% (40 out of 40 attempts)

- **Sampling:** 90.9% (30 out of 33 attempts)
- **Sampling 2:** 88.2% (30 out of 34 attempts)
- **Lineage:** 81.6% (31 out of 38 attempts)
- **Random:** 56.9% (33 out of 58 attempts)
- **Cardinality:** 41.4% (24 out of 58 attempts)

As expected, the **Oracle** achieved a perfect 100% hit rate, serving as the theoretical benchmark. Among the practical heuristics, the **Sampling** strategy proved to be the most reliable predictor, though **Sampling 2** and **Lineage** were not far behind. All three demonstrated a strong ability to make effective choices.

The most striking result is the poor performance of the **Cardinality** strategy. With a hit rate of only 41.4%, it was significantly outperformed by a purely **Random** selection (56.9%). This strongly suggests that for this type of complex, compositional query workload, the optimizer's internal cardinality estimates are not just unreliable but are actually worse than making a random guess among the viable candidates.

The primary reason the Sampling strategy can have a better Hit Ratio than the Lineage strategy, even on uniform data, is that its predictive model is more **statistically robust**.

The Data Lineage strategy is akin to predicting an outcome based on a **single roll of the dice**; it derives its entire forecast from the one, often small and statistically noisy, result of the initial failed query. This makes its prediction brittle and highly sensitive to random variance.

In contrast, the Sampling strategy effectively **rolls the dice multiple times**; by running the query on several different sample sizes, it builds a linear regression model from multiple data points. This process "smooths out" the noise from any single outcome and captures the overall trend, resulting in a more reliable prediction, especially for complex joins where its holistic, end-to-end measurement is a key advantage.

It is important to clarify the difference between the total attempts for different strategies. For example, in our experiments, Sampling tried to find an intelligent re-creation 33 times. The Oracle may have made more attempts overall. This does not mean Sampling performed fewer re-creations in total, but rather that there were scenarios where its prediction model was too conservative to identify a viable candidate, forcing an immediate fallback to the baseline. The Oracle, with its perfect knowledge, could find a solution in those cases.

4.3.2 Metric 2: Total Remote Re-creation Cost (seconds)

This metric is the sum of the remote execution costs, including the 100ms latency penalty, for every re-creation event across all sessions. Additionally, it includes the overhead from fallback scenarios where our strategy selected a bad candidate and the system had to revert to the baseline approach, capturing the real-world cost of suboptimal decisions.

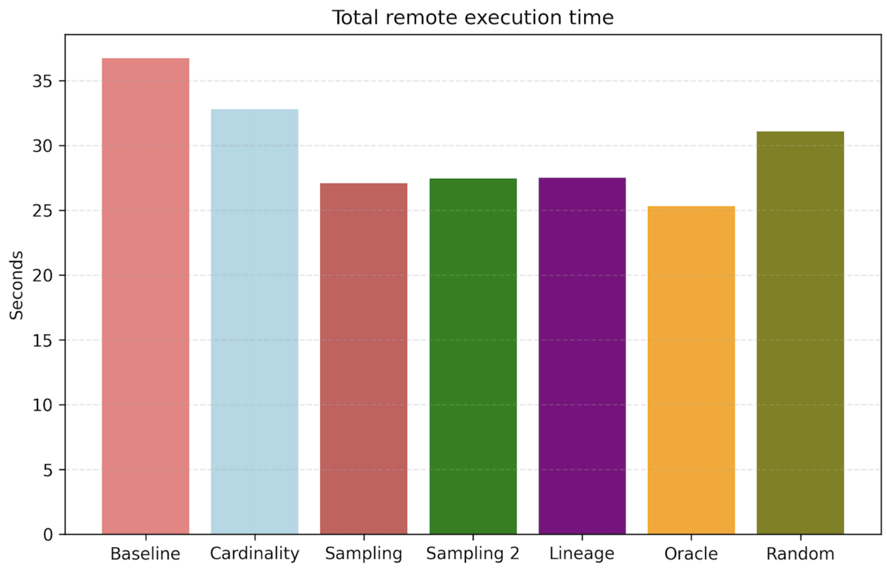


Figure 4.2: Total remote re-creation cost (in seconds) aggregated across all sessions for each strategy.

The final ranking is as follows:

- **Oracle:** 25.31s
- **Sampling:** 27.10s
- **Sampling 2:** 27.45s
- **Lineage:** 27.52s
- **Random:** 31.09s
- **Cardinality:** 32.82s
- **Baseline:** 36.76s

The **Data Lineage**, **Sampling**, and **Sampling 2** strategies were the clear top performers. Their total costs were not only close to each other but also stayed within a reasonable margin of the **Oracle's** theoretical optimum. This is a strong signal that these heuristics are effective at keeping the cumulative user wait time down.

At the other end of the spectrum, the **Baseline** strategy was by far the worst performer at 36.76s — significantly higher than that of the other strategies. This powerfully validates the core premise of this thesis: that an intelligent recovery strategy is critical.

Interestingly, even the flawed heuristics provided a significant improvement over the naive fallback. The **Cardinality** strategy (32.82s), despite its frequent prediction errors, still beat the baseline. This indicates that the cost saved during its occasional correct choices was enough to offset its mistakes. Even more striking is that a purely **Random** selection strategy (31.09s) also comfortably outperformed the baseline, reinforcing the conclusion that almost any targeted re-creation is better than the brute-force approach.

4.3.3 Metric 3: Total Local Execution Time (seconds)

This metric accounts for the client-side computational overhead introduced by each strategy's decision-making process. While client-side computation on in-memory data is generally fast, some of the more complex heuristics may add non-negligible processing time. This metric measures that overhead, aggregated across all sessions.

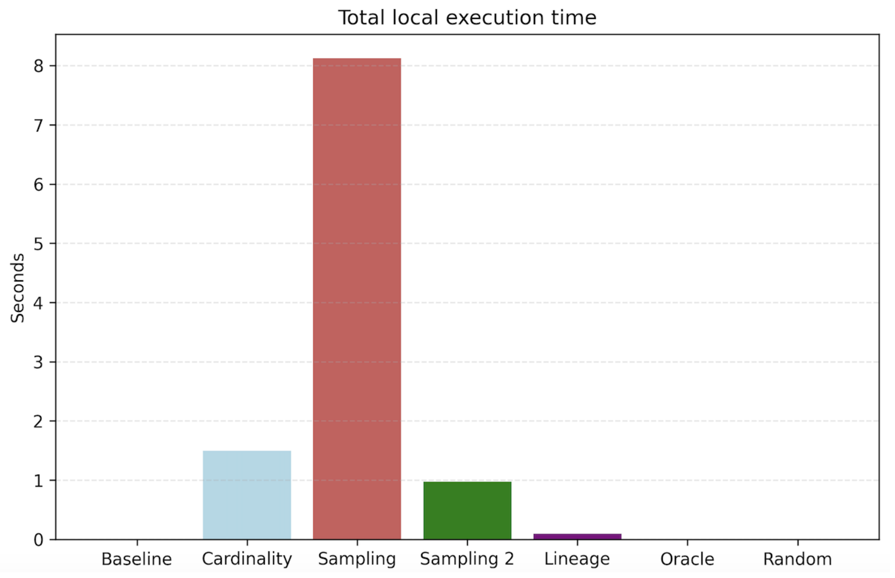


Figure 4.3: Total local execution time (in seconds) introduced by the decision-making logic of each strategy.

The results are as follows:

- **Sampling:** 8.12s
- **Cardinality:** 1.49s
- **Sampling 2:** 0.97s
- **Lineage:** 0.09s
- **Oracle, Random, Baseline:** Negligible (∼0s)

As expected, the **Sampling** strategy was by far the most expensive. This high overhead is a direct consequence of its exhaustive methodology, which requires taking multiple samples (25%, 50%, 75%) for every potential candidate and re-running the query for each one.

In contrast, the **Sampling 2** approach proved to be much more efficient. At 0.97s, its hierarchical approximation method, which only samples immediate children and reuses previously computed slopes, drastically reduces the local workload, making it a more feasible option.

The **Lineage** strategy was exceptionally lightweight, incurring a negligible overhead of only 0.09s. This is because its analysis—calculating selectivity and expansion factors—is a simple operation that is linear in the size of the small failed result set.

The **Cardinality** strategy's overhead of 1.49s stems from the cumulative cost of requesting numerous cardinality estimates from the optimizer for every candidate across all failures. Finally, the **Oracle**, **Random**, and **Baseline** strategies have no significant decision-making logic, and thus their local processing time is effectively zero.

4.3.4 Metric 4: Total Execution Time (seconds)

This metric provides the overall cost of each strategy by summing the **Total Remote Re-creation Cost** and the **Total Local Execution Time**. It represents the complete time overhead—both remote and local client-side processing—introduced by each recovery approach across all sessions.

The final ranking is as follows:

- **Oracle:** 25.31s
- **Lineage:** 27.61s
- **Sampling 2:** 28.42s

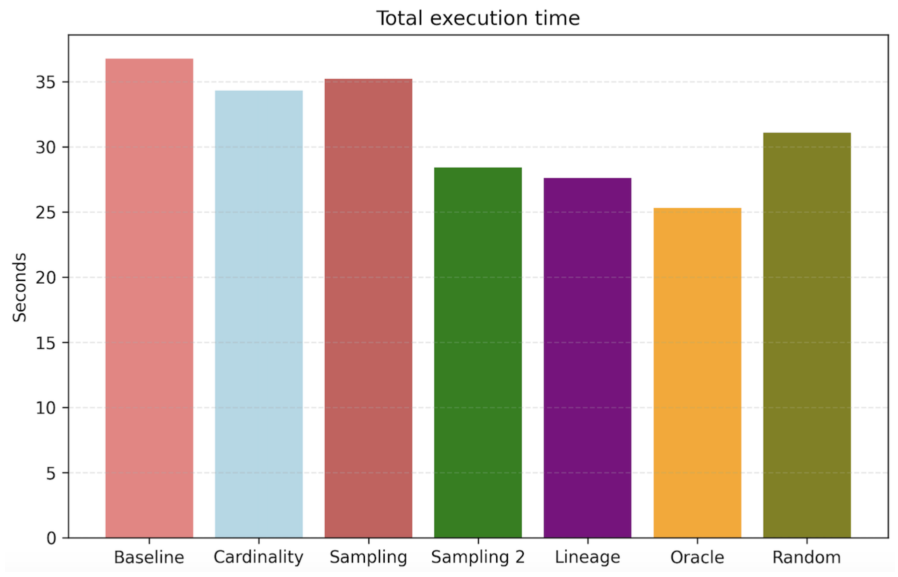


Figure 4.4: Final total execution time (remote + local) for each strategy.

- **Random:** 31.09s
- **Cardinality:** 34.31s
- **Sampling:** 35.21s
- **Baseline:** 36.76s

Among the feasible approaches, the **Data Lineage** strategy emerges as the overall winner, followed closely by the **Sampling 2** strategy. Lineage's victory is due to its superior balance: it achieves a very low remote execution cost while adding almost no local overhead, making it the most efficient and well-rounded solution.

The most notable result here is the dramatic downfall of the exhaustive **Sampling** strategy. Despite its strong predictive accuracy and good remote performance, its extremely high local execution time (8.12s) proved to be its fatal flaw. This overhead was so significant that it performed worse than even the Random and Cardinality approaches, highlighting the critical importance of maintaining a lightweight client-side footprint.

The **Sampling 2** strategy, by effectively reducing this local overhead, proved to be a far more viable approach, securing its place as the second-best heuristic. Finally, the fact that even a pure **Random** approach provided a significant improvement over the **Baseline** confirms that the targeted re-creation framework is fundamentally more efficient than the naive fallback.

4.3.5 Metric 5: Total Number of Re-creations

This metric measures the overall efficiency of each strategy by counting the total number of server-side re-creations triggered across all 20 session files. A lower number suggests that a strategy is more effective at making intelligent choices that enrich the local cache ecosystem, thereby preventing subsequent failures.

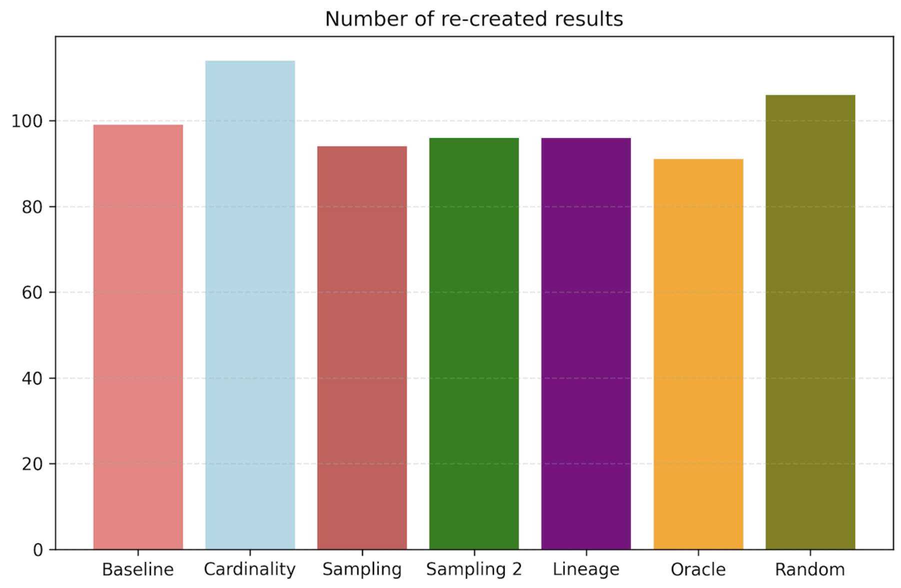


Figure 4.5: Total number of re-creations needed across all 20 sessions for each strategy.

The aggregated results are as follows:

- **Naive Baseline:** 99 re-creations
- **Cardinality Estimation:** 114 re-creations
- **Dynamic Sampling:** 94 re-creations
- **Dynamic Sampling 2:** 96 re-creations
- **Data Lineage:** 96 re-creations
- **Oracle:** 91 re-creations
- **Random:** 106 re-creations

From these results, several key insights emerge. First, the **Cardinality Estimation** approach performed worse than the naive baseline. This is attributed to prediction errors inherent in the optimizer's statistical models. An inaccurate estimate can lead to a suboptimal re-creation choice that fails to satisfy the `MINROWS` constraint, reverting to the baseline method, while the baseline strategy succeeded with a single re-attempt.

In contrast, the **Dynamic Sampling**, **Dynamic Sampling 2** and **Data Lineage** strategies outperformed the baseline, although by a modest margin. Their success stems from a fundamentally different recovery approach. These intelligent heuristics often identify and select `Results` for re-creation that are deeper in the dependency graph—that is, closer to the original base tables.

When such a deep `Result` is re-created, the new tuples trigger a **cascading update** on the client. This process propagates the enrichment upwards through the dependency graph, increasing the number of rows in *all* `Results` that depend on the re-created one. This leads to a more evenly distributed and robust cache ecosystem, rather than the baseline's approach of only enriching the single `Result` that failed. This even distribution is highly effective at preventing future `MINROWS` failures in workloads where various previously created `Results` are re-used throughout the session.

However, in simple, linear sessions where a query only uses the result from the immediately preceding step, the baseline's "brute-force" re-creation can be more effective by concentrating a high number of tuples exactly where they are needed next. The fact that the intelligent strategies' margin of victory was not wide suggests that our synthetic session files had a strong tendency to reuse recently created results, a pattern that partially favors the baseline's simplistic approach.

It is crucial to note that this applies even to the **Oracle** strategy. The Oracle is locally optimal—it always finds the cheapest solution for the *current* failure. However, it does not have future knowledge of which cached results will be most valuable for subsequent queries. Consequently, in linear session patterns, the Baseline's brute-force approach of fully enriching the most recent result can lead to fewer total re-creations over an entire session than the locally-perfect choices made by the Oracle.

Finally, we analyze the total execution time under the cumulative-effect methodology, where the cache state is not restored after a recovery. This metric represents the most holistic measure of performance, as it captures how each strategy's decisions compound over the course of a long session. The total time is the sum of the cumulative remote re-creation costs and the total local processing overhead.

By looking at the results, we see that the most effective strategies remain consistent. The **Data Lineage** and **Sampling 2** approaches still present the best overall performance, demonstrating their strong balance of making intelligent remote choices while maintaining a low client-side footprint.

The most revealing insight from this cumulative analysis comes from the exhaustive **Sampling** strategy. In this scenario, its accurate, data-driven predictions led to a lower total number of re-creations. However, its final total execution time was poor,

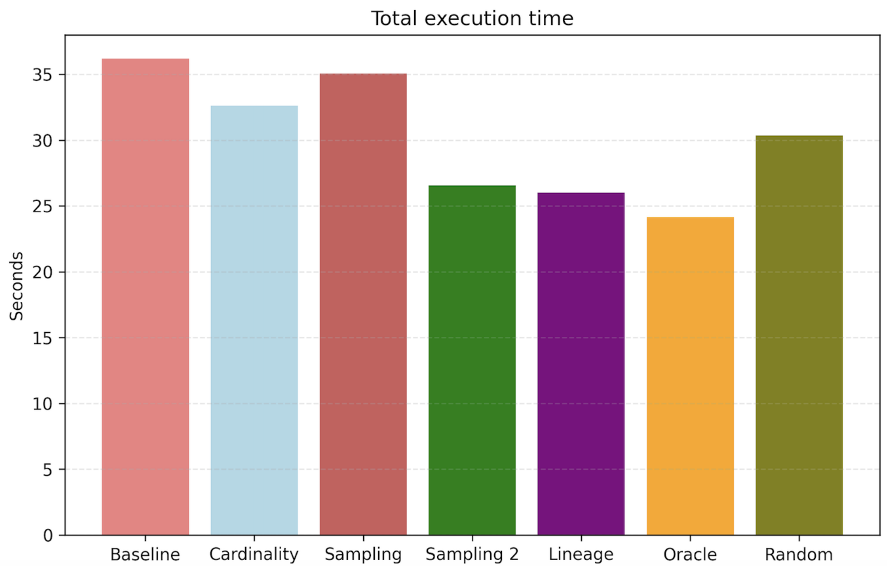


Figure 4.6: Total execution time (remote + local) under the cumulative (no-reset) experimental setup.

ranking even lower than strategies like **Cardinality** which had a much higher number of remote re-creations. This paradox is explained by its local execution cost. The time spent repeatedly running queries on local samples at each decision point ultimately outweighed the time saved from fewer server interactions.

This result powerfully underscores the importance of a lightweight heuristic. For an interactive system, a strategy is only viable if its decision-making process is itself "instant." The Lineage and Sampling 2 approaches succeed because they provide this balance, whereas the exhaustive Sampling strategy, despite its predictive intelligence, fails the test of practical feasibility due to its high computational overhead.

4.3.6 Metric 6: Analysis of Re-creation Selection by Depth

This final metric examines the selection patterns of each strategy by analyzing the distribution of re-creations across different "depths" in the dependency graph. We define **Depth 0** as the top-most `Result` that failed its `MINROWS` constraint. A decision to re-create a `Result` at Depth 0 is therefore equivalent to a **fallback to the naive baseline strategy**. A selection at **Depth 1** represents choosing one of the direct children of the failed `Result`, Depth 2 a "grandchild," and so on.

The goal of this experiment is to see how often each heuristic succeeds in finding an "intelligent" solution (a selection at Depth > 0) versus how often it gives up and resorts to the baseline (a selection at Depth 0).

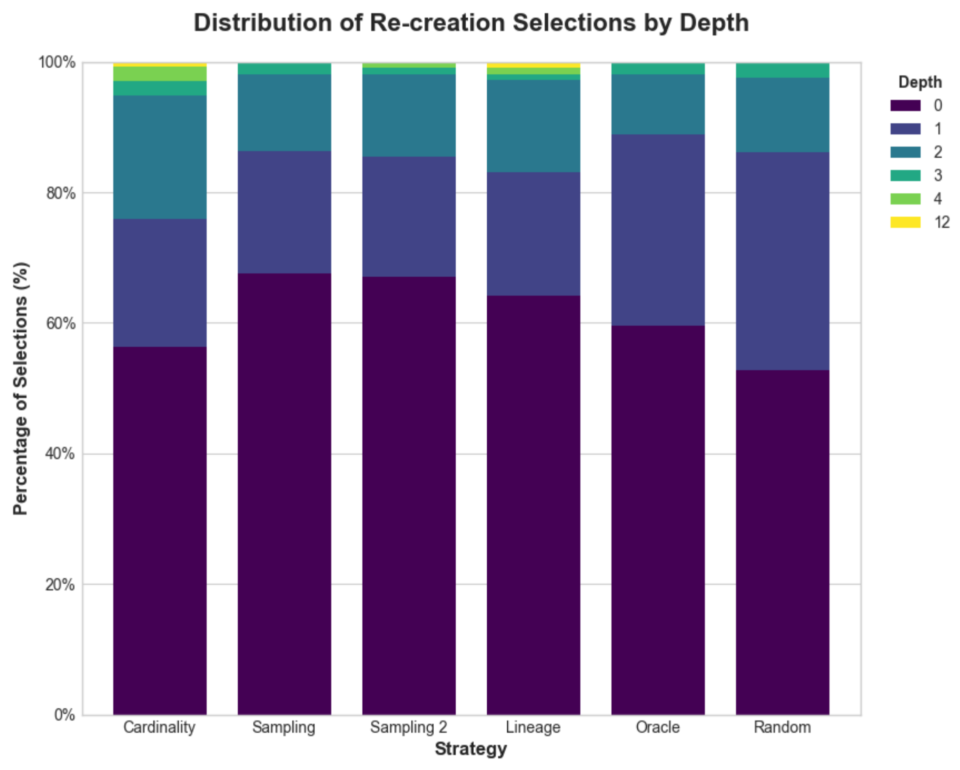


Figure 4.7: Distribution of re-creation selections by depth, where Depth 0 represents a fallback to the baseline strategy.

The data reveals the intelligence and limitations of each strategy. The **Oracle**, with its perfect knowledge, determines that in many instances, the optimal strategy is, in fact, to fall back to the baseline (Depth 0). This establishes a crucial ground truth: in some scenarios, there is no "clever" choice that is better than simply re-computing the final result.

When comparing the practical strategies, we observe different behavioral patterns. The **Data Lineage** and **Cardinality** strategies seem to be more prone to selecting candidates from a deeper depth, whereas the exhaustive **Sampling** and **Sampling 2** strategies appear more conservative in their choices.

This distribution pattern is not just an observation; it can serve as a powerful reference for the development of better heuristics. For example, by noticing that the Oracle strategy's intelligent choices are heavily concentrated at the top levels of the dependency graph, we could introduce a "depth penalty" into a selection model. Such a mechanism would punish the selection of deeper candidates, potentially improving a heuristic by guiding it to favor choices that are more likely to be optimal.

4.4 Evaluation on Skewed and Correlated Data

The experiments presented thus far have been conducted using the standard TPC-H data. A known characteristic of TPC-H is that its data generators produce uniformly distributed data. While this provides a stable baseline for evaluation, it may not reflect the characteristics of real-world datasets, which are often skewed and contain complex correlations. The heuristics we have developed might be sensitive to these data properties.

To test the robustness of our strategies under more challenging conditions, we conducted a second set of experiments using skewed and correlated data.

4.4.1 Methodology and Data Generation with JCC-H

For this evaluation, we utilized the **JCC-H Data Generator**. As described by Boncz, Anatiotis, and Klabe in "JCC-H: Adding Join Crossing Correlations with Skew to TPC-H," JCC-H is a drop-in replacement for the standard TPC-H data generator that introduces realistic skew and join-crossing-correlations into the dataset [BAK17]. This provides a convenient way to assess how a system that is already tested with TPC-H can handle more complex data distributions.

We used the same 20 synthetically generated session files as in the previous experiments, but executed them against the JCC-H dataset. To ensure a fair, head-to-head comparison of how each strategy's prediction model copes with the difficult data, the local cache states were reset after every recovery action to a common recovered state, forcing all strategies to be evaluated on the very same queries that failed to produce enough tuples.

The following sections present the results for four key metrics under these skewed and correlated data conditions.

4.4.2 Metric 1: Hit Ratio on JCC-H Data

This metric isolates the predictive reliability of each strategy when faced with a more realistic dataset containing skew and join-crossing-correlations. We define a "hit" as an event where a strategy selects a candidate for re-creation (other than the baseline), and that choice successfully produces enough tuples to meet the `MINROWS` requirement. The ratio measures how often an "intelligent" choice is a correct one.

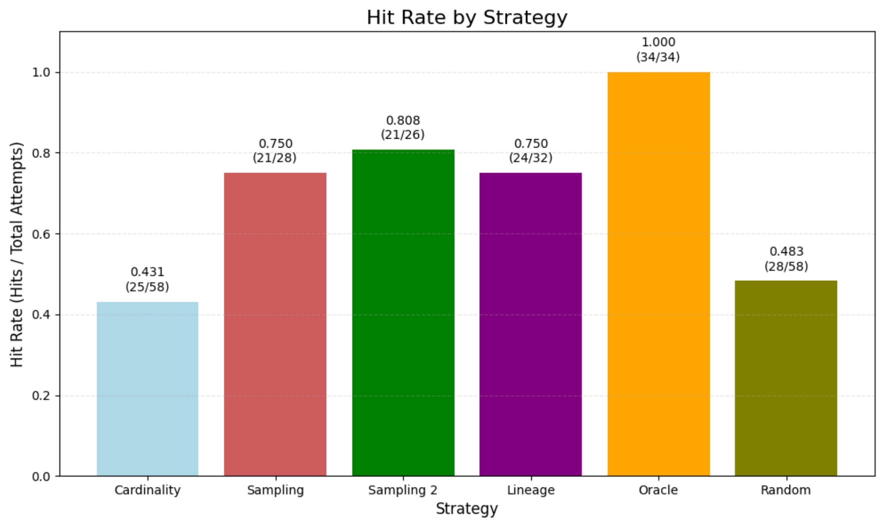


Figure 4.8: Predictive accuracy (Hit Ratio) of each strategy's non-baseline choices on skewed JCC-H data.

The results show a clear hierarchy in predictive accuracy under these more challenging conditions:

- **Oracle:** 100% (34 out of 34 attempts)
- **Sampling:** 75% (21 out of 28 attempts)
- **Sampling 2:** 80.8% (21 out of 26 attempts)
- **Lineage:** 75% (24 out of 32 attempts)
- **Random:** 48.3% (28 out of 58 attempts)
- **Cardinality:** 43.1% (25 out of 58 attempts)

As expected, the **Oracle** achieved a perfect 100% hit rate, serving as the theoretical benchmark. Among the practical heuristics, **Sampling 2** emerged as the most reliable predictor with an 80.8% hit rate, closely followed by **Sampling** and **Lineage**, both at 75%. The strong performance of all three strategies demonstrates their robustness, as they maintained high predictive accuracy even when faced with skewed and correlated data.

The most striking result is the continued poor performance of the **Cardinality** strategy. With a hit rate of only 43.1%, it was once again outperformed by a purely **Random** selection (48.3%).

It is also important to consider the total number of attempts made by each strategy. For example, Lineage attempted an intelligent re-creation 32 times, whereas Sampling 2 only did so 26 times. This indicates that some models are more "conservative" and are quicker to fall back to the baseline if they cannot find a candidate that meets a high confidence threshold. The Oracle, with its perfect knowledge, found the most opportunities (34) for a successful non-baseline fix.

4.4.3 Metric 2: Total Remote Re-creation Cost on JCC-H Data

This metric is the sum of the remote execution costs, including the 100ms latency penalty, for every re-creation event across all sessions. Additionally, it includes the overhead from fallback scenarios where our strategy selected a bad candidate and the system had to revert to the baseline approach, capturing the real-world cost of suboptimal decisions.

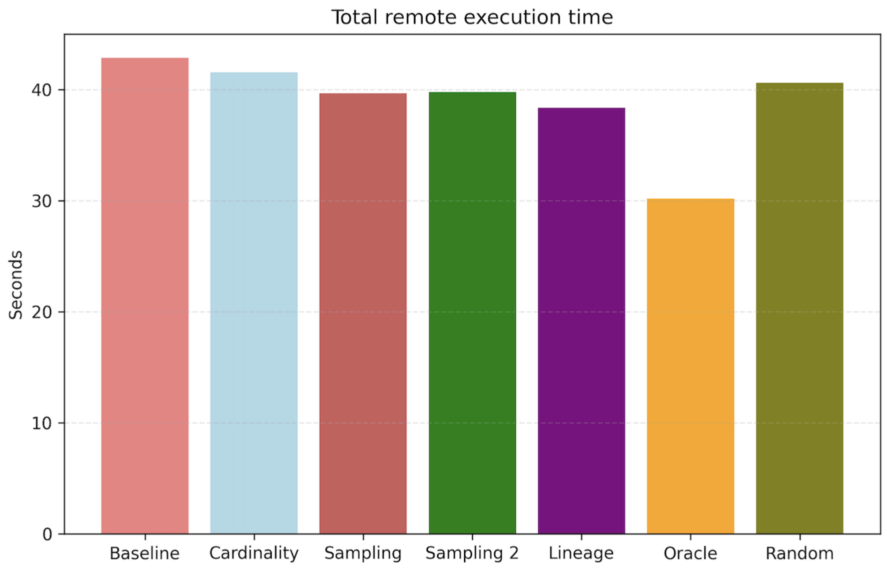


Figure 4.9: Total remote re-creation cost (in seconds) on skewed JCC-H data.

The results show a clear performance hierarchy on the skewed dataset:

- **Oracle:** 30.19s
- **Lineage:** 38.38s
- **Sampling:** 39.67s
- **Sampling 2:** 39.79s

- **Random:** 40.63s
- **Cardinality:** 41.56s
- **Baseline:** 42.88s

The **Lineage**, **Sampling**, and **Sampling 2** strategies were the clear top performers. However, unlike the experiments on uniform data, their total costs were not far ahead of the less effective strategies like Random and Cardinality. Furthermore, a significant performance gap still exists between these top heuristics and the theoretical optimum set by the **Oracle**. This indicates that while they are the best practical options, there is still margin for improvement, and the skewed, correlated data makes it fundamentally more difficult for any heuristic to achieve optimal performance.

The **Cardinality** strategy was the least effective heuristic, reinforcing the findings from the Hit Ratio metric. The fact that a purely **Random** selection strategy performed slightly better again suggests that the optimizer's internal cardinality estimates are not well-suited for this task, especially on skewed data where such estimates are notoriously difficult.

4.4.4 Metric 3: Total Local Execution Time on JCC-H Data

This metric accounts for the client-side computational overhead introduced by each strategy's decision-making process on the skewed dataset.

The results are as follows:

- **Sampling:** 7.37s
- **Cardinality:** 1.43s
- **Sampling 2:** 0.82s
- **Lineage:** 0.08s
- **Oracle, Random, Baseline:** Negligible ($\sim$0s)

Confirming the results from our experiments on uniform data, the pattern of local overhead costs remains consistent. The exhaustive **Sampling** strategy (7.37s) was once again the most expensive by a wide margin due to its computationally intensive methodology. As discussed earlier, the **Lineage** strategy (0.08s) proved exceptionally lightweight, while the **Sampling 2** approach (0.82s) offered a much more feasible performance compromise. The overhead of the **Cardinality** strategy (1.43s) was also in line with previous findings, and the remaining approaches had no significant local cost.

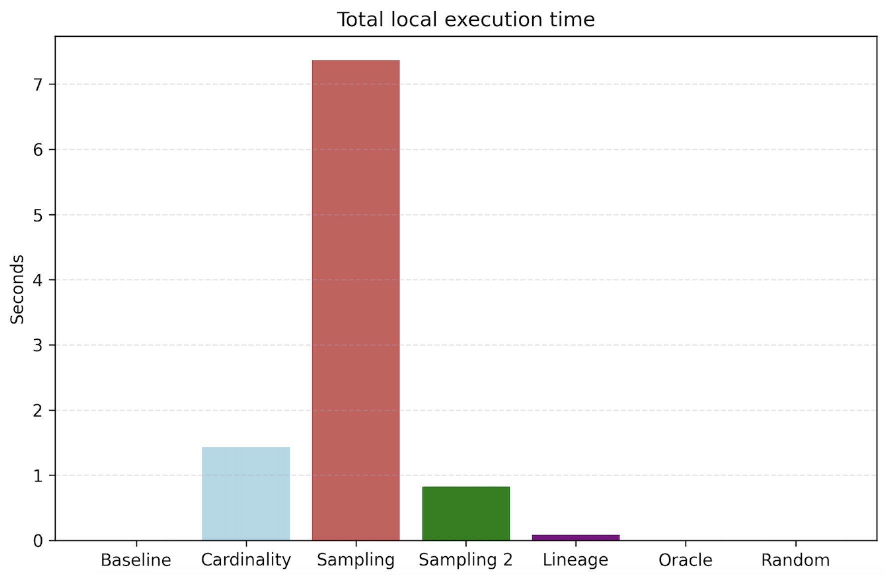


Figure 4.10: Total local execution time (in seconds) introduced by the decision-making logic of each strategy on skewed JCC-H data.

4.4.5 Metric 4: Total Execution Time on JCC-H Data

This metric provides the overall cost of each strategy by summing the **Total Remote Re-creation Cost** and the **Total Local Execution Time** on the skewed dataset. It represents the complete time overhead—both user-facing wait time and client-side processing—introduced by each recovery approach.

The final ranking is as follows:

- **Oracle:** 30.19s
- **Lineage:** 38.46s
- **Sampling 2:** 40.61s
- **Random:** 40.63s
- **Baseline:** 42.88s
- **Cardinality:** 42.99s
- **Sampling:** 47.04s

The **Data Lineage** strategy emerges as the most well-rounded and effective heuristic on the skewed dataset. However, its victory comes with two important caveats. First, the

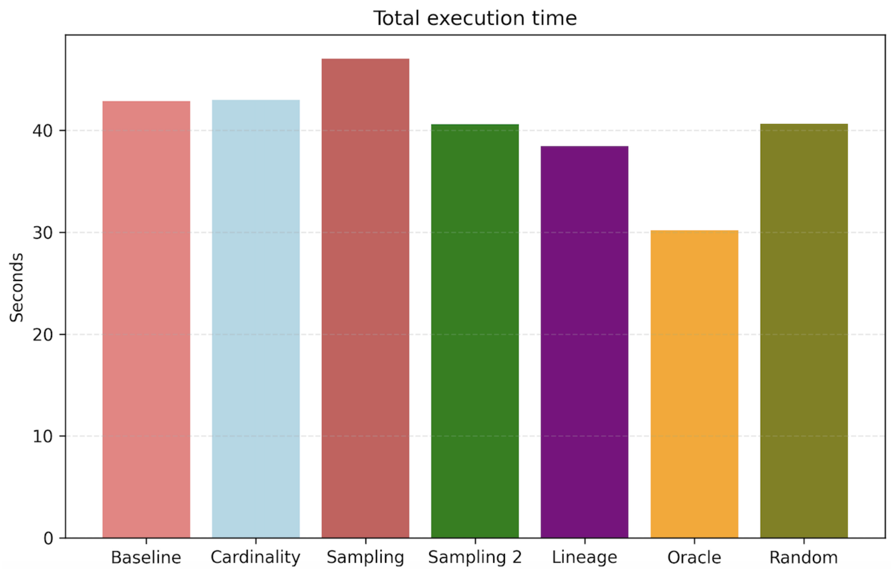


Figure 4.11: Final total execution time (remote + local) for each strategy on skewed JCC-H data.

performance improvement over the **Baseline** is not as great as it was on uniform data. Second, a significant gap remains between Lineage's performance and the theoretical optimum set by the **Oracle**, indicating that skewed data presents a difficult challenge with a clear margin for future improvement.

As discussed in the previous section, the exhaustive **Sampling** strategy's final ranking is exceptionally poor due to its massive local overhead, which completely negated any benefits from its remote re-creation choices. Its approximated counterpart, **Sampling 2**, proved to be a far more viable approach. Its total time was competitive, though notably very close to that of a purely **Random** strategy, highlighting the difficulty of making consistently optimal choices on this dataset.

4.4.6 Analysis of Re-creation Selection by Depth on JCC-H Data

The distribution of re-creation selections by depth on the skewed JCC-H data reveals a fundamentally more challenging environment than that of the uniform TPC-H data.

The most telling observation comes from the **Oracle** strategy. The graph shows that even with perfect knowledge, the Oracle was forced to fall back to the baseline strategy (Depth 0) in the vast majority of cases. This establishes a crucial ground truth: the skewed and correlated nature of the data presented far fewer opportunities for an intelligent, "surgical" re-creation choice to be effective. This directly explains why the

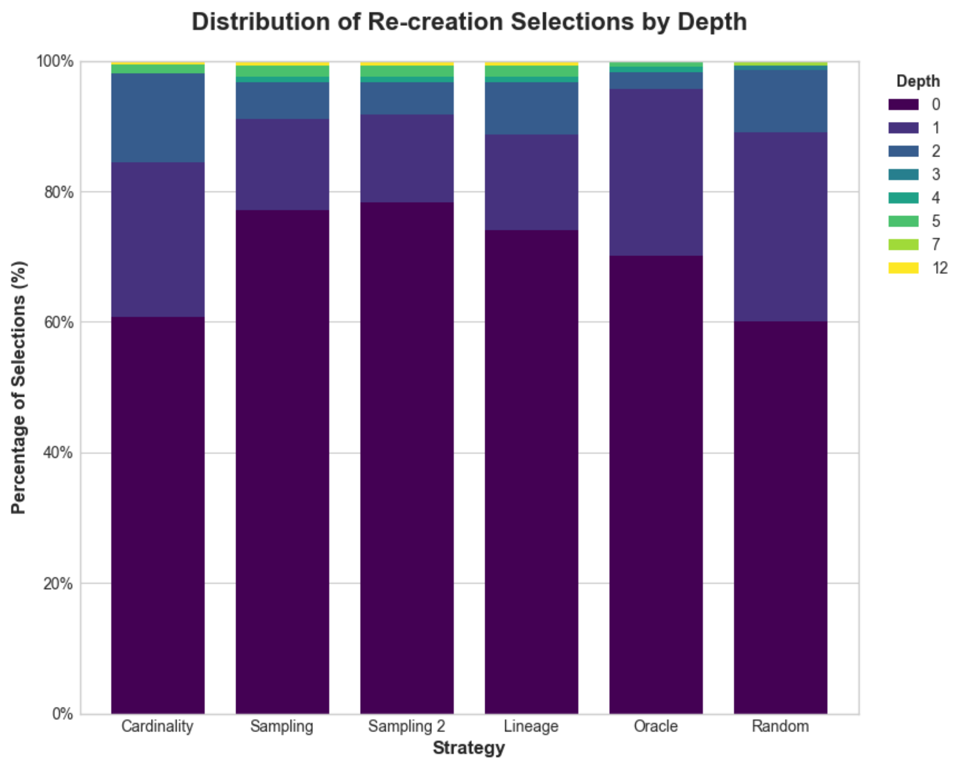


Figure 4.12: Distribution of re-creation selections by depth on skewed JCC-H data, where Depth 0 represents a fallback to the baseline strategy.

performance gains of the best heuristics over the baseline were smaller in this set of experiments.

Furthermore, for the few cases where a non-baseline choice was viable, the selections were heavily concentrated at **Depth 1**. There were very few opportunities to make beneficial re-creations at deeper levels of the dependency graph, which further limited the potential for significant cost savings.

It is important to point out that the JCC-H dataset, while valuable for stress-testing, may represent an extreme level of skew that is not necessarily typical of all production workloads. However, seeing how the strategies behave under these conditions is still a valuable exercise. In reality, we expect the characteristics of most datasets to lie somewhere on the spectrum between the perfectly uniform TPC-H distribution and this highly skewed one.

5 Conclusion

Modern data analysis demands interactive tools that provide immediate feedback, a need addressed by features like MotherDuck's `Instant SQL`. Powering this real-time experience in a hybrid cloud/browser environment requires a sophisticated caching system. This thesis tackled a critical challenge within such systems: how to intelligently recover when a query on partial, cached data fails to produce a sufficiently large result. We proposed and evaluated a "Targeted Re-creation" framework to intelligently recover from `MINROWS` failures. Through extensive experimentation on both uniform and skewed data, we compared three novel heuristics—Cardinality Estimation, Dynamic Sampling, and Data Lineage—against a naive Baseline, an idealized Oracle, and a purely Random approach.

Based on the comprehensive results, our final recommendation is the **Data Lineage** strategy. This approach consistently demonstrated the best overall performance, providing a superior balance of low local execution overhead and efficient remote re-creation choices. It proved to have a strong predictive ability, enabling it to select effective re-creation candidates at a very low cost.

The minimal local overhead of the Lineage strategy is a particularly compelling advantage. The target environment for this feature, DuckDB Wasm, is currently single-threaded, making it imperative to keep the client-side computational load as low as possible to ensure a responsive user interface. The exhaustive Sampling strategy, despite its predictive power, was ultimately disqualified by its high local cost, while the Lineage strategy's overhead was negligible.

Furthermore, the implementation of the Data Lineage strategy is feasible and does not introduce prohibitive complexity. It requires augmenting cached results with an extra metadata column to store sets of origin `Result` identifiers. These sets are simply propagated and unioned as new `Results` are composed. As demonstrated in our analysis, the system only needs to track the source `Results` for each tuple, not the specific source row indices, which significantly simplifies the mechanism and reduces the metadata footprint.

Finally, it is crucial to emphasize that regardless of which intelligent heuristic is chosen, the **Naive Full Replacement (Baseline)** strategy must be implemented alongside it. It serves as the indispensable fallback mechanism for the two scenarios where a targeted fix is not possible: when a chosen re-creation candidate fails to produce enough rows, or when the heuristic cannot identify any viable candidate in the first place.

5.0.1 Future Work

The research presented in this thesis establishes a robust framework for managing compositional caches, but it also opens up several exciting avenues for future investigation.

Result Stability in Interactive Workflows This work focused on ensuring a *sufficient* number of rows, but not on the *stability* of those rows between incremental query edits. In an `Instant SQL` context, it is desirable for the tuples a user sees in a preview to remain present and in a similar order as they refine their query. Future work could introduce and evaluate metrics to measure this, such as **result containment** or **top-k stability**. One could also explore recovery mechanisms that explicitly preserve stability, for instance, by performing a `UNION` of the old, small result set with the new, re-created one to guarantee that previously visible tuples are not lost.

Bidirectional and Server-Side Caching The current model treats `Result` caches as a client-only phenomenon. A possible direction would be to make the server aware of these caches, creating a bidirectional caching system. If the server maintains its own copies of `Results`, it could open up significant optimization opportunities for hybrid queries that mix base tables and cached results. Such queries could be executed entirely and efficiently on the server side, eliminating the need to upload client data and enabling a broader class of queries to be accelerated.

Heuristic Refinement from Experimental Insights The experimental results themselves can be used to develop more refined heuristics. The analysis of the re-creation selections by depth, for instance, showed that the Oracle's optimal choices were heavily concentrated at shallow depths. This insight could be used to improve other strategies. For example, one could introduce a "depth penalty" into the selection model, which would programmatically punish the selection of deeper `Results`, thereby guiding a heuristic's behavior to be more similar to the Oracle's without overfitting to a specific workload.

Validation on Real-World Datasets While the use of LLM-generated workloads on TPC-H and JCC-H data provided a controlled and challenging experimental environment, the final step would be to validate these findings on a large, real-world dataset. Using a public dataset, such as the Stack Overflow data dump [14b; 14a], would allow us to evaluate how the strategies behave when faced with the organic skew, correlations, and complexity of production data, further strengthening the conclusions of this work.

Abbreviations

AQP Approximate Query Processing

SQL Structured Query Language

TPC-H Transaction Processing Performance Council - H

JCC-H Join Crossing Correlations - H

LLM Large Language Model

RDBMS Relational Database Management System

DBMS Database Management System

DBA Database Administrator

OLAP Online Analytical Processing

WASM WebAssembly

WAN Wide Area Network

List of Figures

Bibliography

[14a] *Stack Exchange Blog Post*. 2014. URL: https://stackoverflow.blog/2014/01/23/stack-exchange-cc-data-now-hosted-by-the-internet-archive/ (visited on 2014).

[14b] *Stack Exchange Data*. 2014. URL: https://archive.org/details/stackexchange (visited on 2024).

[21] *SQLGlot*. 2021. URL: https://sqlglot.com/sqlglot/executor.html (visited on 01/01/2025).

[25a] *30 000 Stars on GitHub*. 2025. URL: https://duckdb.org/2025/06/06/github-30k-stars.html.

[25b] *DuckDB Python Library*. 2025. URL: https://duckdb.org/docs/stable/clients/python/overview.html (visited on 01/01/2025).

[25c] *Instant SQL*. 2025. URL: https://motherduck.com/blog/introducing-instant-sql (visited on 01/01/2025).

[25d] *MotherDuck Documentation*. 2025. URL: https://motherduck.com/docs/ (visited on 01/01/2025).

[25e] *WebAssembly*. 2025. URL: https://webassembly.org/ (visited on 01/01/2025).

[AA23] A. Author and B. Author. "MotherDuck: Hybrid Query Execution in the Browser." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2023), pp. 1–12.

[Aga+13] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data." In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. 2013, pp. 29–42.

[BAK17] P. Boncz, A.-C. Anatiotis, and S. Klabe. "JCC-H: adding join crossing correlations with skew to TPC-H." In: *Proceedings of the International Workshop on Testing Database Systems*. 2017, pp. 1–6.

[BKT01] P. Buneman, S. Khanna, and W.-C. Tan. "Why and Where: A Characterization of Data Provenance." In: *Lecture Notes in Computer Science* 1997 (2001), pp. 316–330.

[BZN05] P. Boncz, M. Zukowski, and N. Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *Proceedings of International conference on verly large data bases (VLDB) 2005*. Jan. 2005.

[Cod70] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks." In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[Cra+15] K. Cranmer, L. Heinrich, R. Jones, D. M. South, and for the ATLAS collaboration. "Analysis Preservation in ATLAS." In: *Journal of Physics: Conference Series* 664.3 (Dec. 2015), p. 032013. DOI: 10.1088/1742-6596/664/3/032013.

[Dai+08] C. Dai, D. Lin, E. Bertino, and M. Kantarcioglu. "An Approach to Evaluate Data Trustworthiness Based on Data Provenance." In: *Secure Data Management*. Ed. by W. Jonker and M. Petković. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 82–98. ISBN: 978-3-540-85259-9.

[DG15] B. Dietrich and T. Grust. "A SQL Debugger Built from Spare Parts: Turning a SQL: 1999 Database System into Its Own Debugger." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 865–870. ISBN: 9781450327589. DOI: 10.1145/2723372.2735358.

[Gra93] G. Graefe. "Query Evaluation Techniques for Large Databases." In: *ACM Computing Surveys* 25.2 (1993), pp. 73–170.

[HH99] P. J. Haas and J. M. Hellerstein. "Ripple Joins for Online Aggregation." In: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. 1999, pp. 287–298.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. "Online Aggregation." In: *SIGMOD Record* 26.2 (1997), pp. 171–182.

[IC91] Y. E. Ioannidis and S. Christodoulakis. "On the Propagation of Errors in the Size of Join Results." In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. 1991, pp. 268–277.

[Iva+09] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. "An architecture for recycling intermediates in a column-store." In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: Association for Computing Machinery, 2009, pp. 309–320. ISBN: 9781605585512. DOI: 10.1145/1559845.1559879.

[KB96] A. M. Keller and J. Basu. "A Predicate-based Caching Scheme for Client-Server Database Architectures." In: *The VLDB Journal*. Vol. 5. 1. 1996, pp. 35–47.

[Lei+15] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. "How good are query optimizers, really?" In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 204–215. ISSN: 2150-8097. DOI: `10.14778/2850583.2850594`.

[NA23] A. Niclas and B. Author. "Declarative Caching for Interactive Data Applications." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2023), pp. 1–15.

[Par+18a] Y. Park, B. Mozafari, J. M. Hellerstein, and M. J. Franklin. "VerdictDB: Universal Approximate Query Processing." In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 2018, pp. 491–506.

[Par+18b] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. "VerdictDB: Universalizing Approximate Query Processing." In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 1461–1476. ISBN: 9781450347037. DOI: `10.1145/3183713.3196905`.

[Poo+96] V. Poosala, P. J. Haas, Y. E. Ioannidis, and E. J. Shekita. "Improved Histograms for Selectivity Estimation of Range Predicates." In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. 1996, pp. 294–305.

[RM17] M. Raasveldt and H. Mühleisen. "Don't hold my data hostage: a case for client protocol redesign." In: *Proc. VLDB Endow.* 10.10 (June 2017), pp. 1022–1033. ISSN: 2150-8097. DOI: `10.14778/3115404.3115408`.

[RM19] M. Raasveldt and H. Mühleisen. "DuckDB: an Embeddable Analytical Database." In: *Proceedings of the 2019 International Conference on Management of Data*. ACM. 2019, pp. 1981–1984. DOI: `10.1145/3299869.3320212`.

[Roy+00] P. Roy, S. Seshadri, S. Sudarshan, and S. A. Bhobe. "Exchequer: A Caching System for Data Warehouses." In: *Proceedings of the 16th International Conference on Data Engineering (ICDE)*. 2000, pp. 701–704.

[Sch+25] T. Schmidt, V. Leis, P. Boncz, and T. Neumann. "SQLStorm: Taking Database Benchmarking into the LLM Era." In: *Proc. VLDB Endow.* 18.11 (2025), pp. 4144–4157.

[Sel+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. "Access Path Selection in a Relational Database Management System." In: *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (1979), pp. 23–34.

[WE14a] A. Watt and N. Eng. *Chapter 7: The Relational Data Model*. 2014. URL: `https://opentextbc.ca/dbdesign01/chapter/chapter-7-the-relational-data-model/`.

[WE14b] A. Watt and N. Eng. *SQL (Structured Query Language)*. 2014. URL: `https://opentextbc.ca/dbdesign01/chapter/sql-structured-query-language/`.