



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

**SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS**

BSc THESIS

**Parallelizing control flow in mixed imperative - SQL
analytics using speculation**

Evangelos G. Danias

**Supervisors: Dimitrios Gunopoulos, Professor (NKUA)
Anastasia Ailamaki, Professor (EPFL)**

ATHENS

JULY 2022



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

**Παραλληλισμός ελέγχου ροής σε μεικτά
προστακτικά-SQL προγράμματα ανάλυσης δεδομένων
χρησιμοποιώντας εικασίες**

Ευάγγελος Γ. Δανιάς

**Επιβλέποντες: Δημήτριος Γουνόπουλος, Καθηγητής (ΕΚΠΑ)
Αναστασία Αϊλαμάκη, Καθηγήτρια (ΕΡΦΛ)**

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2022

BSc THESIS

Parallelizing control flow in mixed imperative - SQL analytics using speculation

Evangelos G. Danias

S.N.: 1115201800039

SUPERVISORS: **Dimitrios Gunopoulos**, Professor (NKUA)
Anastasia Ailamaki, Professor (EPFL)

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Παραλληλισμός ελέγχου ροής σε μεικτά προστακτικά-SQL προγράμματα ανάλυσης
δεδομένων χρησιμοποιώντας εικασίες

Ευάγγελος Γ. Δανιάς

A.M.: 1115201800039

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Δημήτριος Γουνόπουλος, Καθηγητής (ΕΚΠΑ)**
Αναστασία Αϊλαμάκη, Καθηγήτρια (EPFL)

ABSTRACT

Data analysis in the present day is moving at breakneck speed, with an ever increasing amount of companies and organizations abandoning the structured query languages in favor of mixed imperative-SQL workflows.

The engines that execute these mixed programs, however, are currently not capable of resolving dependencies between queries and the imperative constructs (e.g. control flow dependencies), thus commonly adopting an (almost) query-at-a-time execution fashion which heavily limits task-parallelism. Instead, the available resources are allocated in order to improve data parallelism, which can quickly lead to diminishing returns depending on the nature of the workflow being executed.

In this thesis, we propose a unified architecture which bridges the code parsing and execution with the analytical processing engine. The synergy between these two components allows the OLAP engine to become code-aware, thus unlocking many opportunities of parallelizing queries that would otherwise remain unexploited. Building upon this architecture, we develop a paradigm that relaxes control-flow dependencies and increases task parallelism, a strategy that was not able to prosper with the current engine architecture.

SUBJECT AREA: Intersection of Programming Language Theory & Data-Intensive Systems

KEYWORDS: speculation, dependencies, SQL, imperative programming, parallelism

ΠΕΡΙΛΗΨΗ

Η ανάλυση δεδομένων στις μέρες μας προχωρά με ιλιγγιώδη ταχύτητα, με έναν ολοένα αυξανόμενο αριθμό εταιρειών και οργανισμών να εγκαταλείπουν τις δομημένες γλώσσες ερωτημάτων υπέρ των μεικτών ροών εργασίας προστακτικού-SQL προγραμματισμού.

Ωστόσο, τα συστήματα που εκτελούν αυτά τα μικτά προγράμματα δεν είναι επί του παρόντος ικανά να επιλύσουν τις εξαρτήσεις μεταξύ των ερωτημάτων-εντολών και των προστακτικών δομών (π.χ. εξαρτήσεις ροής ελέγχου), υιοθετώντας έτσι συνήθως έναν (σχεδόν) σειριακό τρόπο εκτέλεσης εντολών το οποίο περιορίζει σε μεγάλο βαθμό τον παραλληλισμό των έργων. Αντίθετα, οι διαθέσιμοι πόροι κατανέμονται προκειμένου να βελτιωθεί ο παραλληλισμός δεδομένων, ο οποίος μπορεί γρήγορα να οδηγήσει σε μειωμένες αποδόσεις ανάλογα με τη φύση της εργασίας που εκτελείται.

Σε αυτή τη εργασία, προτείνουμε μια ενοποιημένη αρχιτεκτονική που γεφυρώνει την εκτέλεση προστακτικού κώδικα με τη μηχανή ανάλυσης των δεδομένων. Η συνέργεια μεταξύ αυτών των δύο συνιστωσών επιτρέπει στο σύστημα ανάλυσης δεδομένων (OLAP) να αποκτήσει γνώση του προγράμματος, ξεκλειδώνοντας έτσι πολλές ευκαιρίες παραλληλισμού ερωτημάτων-εντολών που διαφορετικά θα παρέμεναν ανεκμετάλλευτες. Βασιζόμενοι σε αυτήν την αρχιτεκτονική, αναπτύσσουμε ένα σκελετό επεξεργασίας που χαλαρώνει τις εξαρτήσεις ελέγχου-ροής και αυξάνει τον παραλληλισμό εργασιών, μια στρατηγική που δεν μπόρεσε να ευημερήσει με την τρέχουσα αρχιτεκτονική των αντίστοιχων συστημάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τομή μεταξύ Θεωρίας Γλωσσών Προγραμματισμού και Συστημάτων Ανάλυσης Μεγάλων Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: εικασία, εξαρτήσεις, SQL, προστακτικός προγραμματισμός, παραλληλισμός

ACKNOWLEDGEMENTS

The ideas in this thesis were born and flourished during my Summer 2021 @ EPFL internship in the Data-Intensive and Applications Systems (DIAS) Lab. I am wholeheartedly grateful to my supervisors and collaborators Panagiotis Sioulas and Giagkos Mytilinis, who's excellent assistance and guidance throughout the entire process played a pivotal role in my academic research baptism of fire.

CONTENTS

1. INTRODUCTION	11
1.1 Modern Data Analysis Workflows	11
2. BACKGROUND	13
2.1 Flavors of Parallelism in DBMSs	13
2.1.1 Data vs Task Parallelism	13
2.2 Interpreter Design Run-down	13
2.2.1 Lifecycle Components	13
2.2.2 Symbol Table	14
2.2.3 AST and the Visitor Pattern	14
2.3 Dependencies in Analytical Workflows	15
2.3.1 Data vs Control Dependencies	15
2.3.2 Representing and Scheduling Dependent Tasks	16
2.4 Speculation	16
2.4.1 Branch Prediction	16
2.4.2 Applications in Databases	16
2.5 Approximate Query Processing	17
3. Unified Engine Overview	18
3.1 Prototype Language	18
3.1.1 Outline	18
3.1.2 Declaring Variables & Queries	18
3.1.3 Control Flow Statements	19
3.2 Front End: Parsing	20
3.3 Middle End: Speculative Execution	20
3.3.1 Relaxing Control Flow Dependencies	20
3.3.2 Repairing Mispredictions	21
3.4 Back End: Interpreting & Scheduling Work	22
4. EVALUATION	23
4.1 Setup	23
4.2 Benchmarks	23
5. CONCLUSIONS AND FUTURE WORK	25
ABBREVIATIONS - ACRONYMS	26

LIST OF FIGURES

2.1	Data Dependency of Listing 1.1	15
2.2	Control Dependency of Listing 1.1	15
3.1	Execution flow of Figure 1.1 snippet	20
3.2	Relaxed execution flow of Figure 1.1 snippet	21
4.1	Execution time of all correct predictions vs Non-Speculative version	24
4.2	Execution time of all incorrect predictions vs Non-Speculative version	24

1. INTRODUCTION

1.1 Modern Data Analysis Workflows

Imperative languages such as Python3 and R have seen a huge rise in popularity, especially for exploratory data analysis applications. The rich ecosystem of frameworks and libraries surrounding these languages - such as Pandas and Matplotlib - have made extracting and visualizing useful insights from big data easier than ever. Performing the same analysis through an SQL language would require maintaining and extending a monolithic query, which soon becomes infeasible.

By adding support for injected SQL statements in the imperative paradigm, it is now possible to divide the analysis in small (but now manageable) queries. These queries perform data intensive work, and are thus treated as blackboxes and executed by a back-end OLAP system. The corresponding intermediate results are returned to the front-end parsing / interpreting mechanism and stored in variables. Then, the analyst can perform complex tasks by using these variables in constructs such as loops or control-flow statements.

Thus, it is common to come across code such as in Listing 1.1:

```
Q0 = SQL('SELECT AVG(col) FROM ... WHERE ... ');
y = 100.0;

...

IF Q0 > y THEN
  Q1 = SQL('CREATE TABLE ...');
ELSE
  Q2 = SQL('SELECT col2 FROM ...');
  Q3 = SQL('SELECT SUM(col3) FROM ... WHERE col2 = %q', Q2);
```

Listing 1.1: Sample "Under the Hood" Modern Data Analysis Workflow

We can observe that what would be a monolithic query with multiple levels of nested subqueries, **CASE WHEN** statements etc, can now be translated in the snippet above that is magnitudes order easier to comprehend, maintain and extend.

While the mixed workflow has proven more expressive and powerful than the counterpart structured query languages, the added complex constructs do not come without cost. More specifically, besides the known data dependencies between queries, such as an inner nested subquery needing to fully materialize before the outer one can be evaluated, the imperative constructs also impose new restrictions that can hinder execution.

These dependencies heavily limit the ability of parallelization, if no further action is taken. That is, the established engines currently do not employ strategies of resolving these limitations in order to increase task parallelism opportunities; Instead, they plan and execute one query at a time - agnostic to the rest of the code and program state - while pinning their hopes in increasing data parallelism.

It is trivial to observe that with these design choices, diminishing returns can appear quickly if the workload of the program cannot scale properly, as the extra resources cannot further accelerate the execution of complex queries. For instance, in the Listing 1.1 snippet, the execution time of the **IF/ELSE** statement is largely dependent on the **Q0** latency. Thus, if **Q0** does not scale properly by its nature (e.g. is a **JOIN** heavy query), the engine does not benefit from allocating additional resources to it (data parallelism). This in turn hurts concurrency as the program execution is "blocked" until the branch condition is fully evaluated, while the resources largely remain idle.

We argue that these limitations originate from the current engine architecture, that has decoupled the code execution component from the OLAP system, making it impossible for the engine to have global information about the code and make truly optimal choices. The goal of this work is to explore ways in which we can boost task parallelism, when data parallelism combined with the unresolved control-flow dependencies hinder execution. We believe that this will be achieved by unifying the (currently) separated components, thus enabling all sorts of known optimizations in both the fields of compilers (e.g. Dead Code Elimination) and databases (e.g. Data Sharing between queries). Then, we will employ a novel speculation technique that will relax control flow dependencies, thus unlocking new parallelization opportunities.

2. BACKGROUND

2.1 Flavors of Parallelism in DBMSs

2.1.1 Data vs Task Parallelism

Modern main-memory analytical DBMSs continuously strive to maximize processing throughput by harnessing the power of the underlying (highly parallel) hardware. A batch of concurrent queries can be processed in two fashions:

Data / Intra-Query Parallel: The available processing resources are allocated such that the DBMS executes the operations of a single query in parallel, thus decreasing the latency for long-running queries, especially if they are naturally scalable.

Task / Inter-Query Parallel: The available processing resources are allocated such that the DBMS executes different queries in parallel. If the queries are independent, then the overall performance is improved in contrast to Data / Intra-Query Parallelism DBMSs. However, if the queries form dependencies with each other (e.g updating the same table concurrently), more complicated conflicts arise that reduce the system's throughput.

2.2 Interpreter Design Run-down

2.2.1 Lifecycle Components

A typical interpreter can be divided in multiple distinct components that collaborate in order to take a program as input and output the result. These are explained in order:

- Lexer: The imperative program is read in string form and the source code is turned into a stream of tokens. This term is actually a shortened version of "lexical analysis"; A token is essentially a representation of each item in the code, along with information about its position, line number etc.
- Parser: The tokens are conformed to the appropriate grammar rules in the order they arrive, generating the equivalent nodes for the abstract syntax tree (explained in detail at the end of the section).
- Semantic Analyzer: The only errors that could be caught until this phase are grammar related; Thus, before moving on to interpreting the source code, the AST will be semantically examined such that no rule of the language is violated; For example, in the event of a binary expression e.g addition between two variables, the semantic analyzer will check that their types are eligible to partake in the operation.
- Optimizer: The AST is optimized - while preserving the semantics - in a collections of "passes" over it, each with a unique goal. For example, a pass could remove "dead code" by removing all the unnecessary code which is never going to be executed.

2.2.2 Symbol Table

The symbol table is a data structure used by both compilers and interpreters, where all identifiers (symbols), constants, procedures and functions in a program's source code are associated with information relating to their declaration or appearance in the source. In other words, the entries of a symbol table store the information related to the entry's corresponding symbol, such as:

- The names of all entities in a structured form at one place.
- The mapping of a name to its type and value
- The scope of a name (scope resolution).

That information is utilized at multiple phases of the compilation or interpretation. For example, in the semantic analysis phase, it is most commonly accessed in order to check if a variable has been declared and to verify the correctness of assignment statements & expressions by retrieving and comparing all the types of the variables involved.

2.2.3 AST and the Visitor Pattern

The abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

We also want to traverse an AST for many different purposes; We may want to print the AST, perform semantic analysis, or generate code. Each of these could be accomplished by refining the notion of tree traversal in extensions of some common superclass.

The Visitor pattern was introduced to address the above scenario. Instead of spreading all the code for a given traversal throughout the nodes' classes, the code is concentrated in a particular traversal class. That code is called by arranging for each node to:

- Accept a call from a visitor that performs the traversal
- Call the visitor back using a method in that visitor that is customized to the node

2.3 Dependencies in Analytical Workflows

2.3.1 Data vs Control Dependencies

Consider Listing 1.1; One can notice dependencies forming between queries such as **Q2** and **Q3**, that require **Q2** to fully materialize before **Q3** can evaluate its WHERE clause (Data Dependency):

```

Q0 = SQL('SELECT AVG(col) FROM ... WHERE ... ');
y = 100.0;

...

IF Q0 > y THEN
  Q1 = SQL('CREATE TABLE ...');
ELSE
  Q2 = SQL('SELECT col2 FROM ...');
  Q3 = SQL('SELECT SUM(col3) FROM ... WHERE col2 = %q', Q2);

```

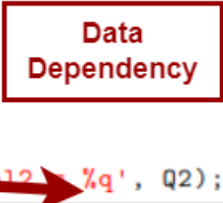


Figure 2.1: Data Dependency of Listing 1.1

If the analysis was performed in a monolithic SQL query, **Q2** would be a nested subquery of **Q3**, but the mixed workflow allows the analyst to simplify the code base by breaking the work apart and/or potentially reuse **Q2**'s result.

Besides that, we also observe that dependencies can now also form between queries and imperative constructs; such example is **Q0**, that must be fully executed before the branch condition is evaluated and consequently any of the **IF/ELSE** branches begin executing (Control Dependency):

```

Q0 = SQL('SELECT AVG(col) FROM ... WHERE ... ');
y = 100.0;

...

IF Q0 > y THEN
  Q1 = SQL('CREATE TABLE ...');
ELSE
  Q2 = SQL('SELECT col2 FROM ...');
  Q3 = SQL('SELECT SUM(col3) FROM ... WHERE col2 = %q', Q2);

```

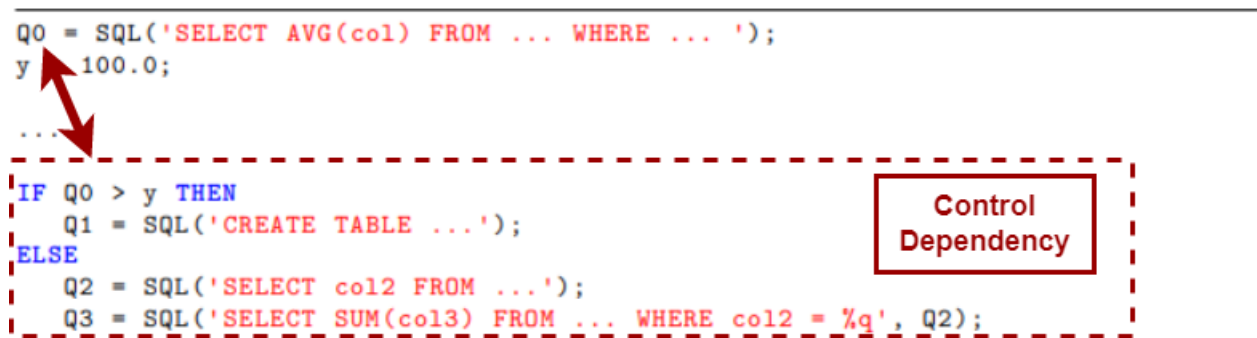


Figure 2.2: Control Dependency of Listing 1.1

2.3.2 Representing and Scheduling Dependent Tasks

Representing all the possible dependence relationships of an imperative program is best done using graph notation. A program dependence graph (PDG) is a directed acyclic graph (DAG) that makes data and control dependencies explicit, thus capturing the flow of execution. In this work, the convention that B is dependent on A if the vertex A is connected to the vertex B ($A \rightarrow B$) is followed.

As the edges on the PDG represent scheduling constraints, it is vital to make sure the task nodes are performed only when all their predecessors are completed. For this matter, one can make use of the topological sort algorithm which produces the natural ordering of the vertices in a directed acyclic graph $G = (V, E)$ by visiting all vertices v before vertices q , iff:

$$v \rightarrow q, \text{ where } v, q \in V, (u, v) \in E$$

2.4 Speculation

In computer science, speculative execution is an established optimization technique in which a computer system performs some task that may not be needed, so as to prevent a delay that would have to be incurred by starting the work after it is known that is needed. If extra resources are available at a certain time, the system can utilize them to perform speculative work instead of allowing them to be idle.

2.4.1 Branch Prediction

Predictive execution is a form of speculative execution where some outcome is predicted and execution proceeds along the predicted path until the actual result is known; If the prediction is true, the predicted execution is allowed to commit; however, if there is a misprediction, execution has to be unrolled and re-executed.

When applied to imperative constructs, the predictive execution of if-then-else statements (branch predictor) attempts to guess which way a branch will go before this is known definitely. This has proven to be of utmost importance in modern hardware, as it keeps feeding work on what would otherwise be a blocked pipeline, thus increasing resource utilization and overall performance.

2.4.2 Applications in Databases

There is a wide variety of speculative techniques employed in the field of databases across many areas. For instance, when an application is accessing pages in a sequential pattern, it is a common procedure for the database system to prefetch the following pages in expectation that they will soon be needed by the application. In addition, in transactional systems where concurrency control is of critical importance, a speculative method of optimistic locking [6] can reduce the expense of managing parallel reads/writes by not locking resources preemptively, thus increasing performance and concurrency opportunities. In relevant work, Sioulas et al [4] showed how complex analytical workflows can be accelerated, by relaxing inter-dependent queries using speculation.

2.5 Approximate Query Processing

Approximate Query Processing (AQP) is a technique that provides approximate answers to queries at a fraction of the time cost. It can do so by executing the query on a statistical sample of the original data, enabling it to finish magnitudes order faster than the initial query, albeit returning a probabilistic answer within a confidence interval. As data grows at an exponential rate with no apparent solution in taming it, the field of AQP is gaining more and more traction as shown in the BlinkDB query engine by Agarwal et al [2], which allows users to trade-off query accuracy (within an error margin) for response time.

3. UNIFIED ENGINE OVERVIEW

3.1 Prototype Language

3.1.1 Outline

In order to fully control how the language constructs will behave with the addition of speculation, we begin by developing a mixed imperative-SQL language with syntax similar to the Listing 1 snippet.

Initially, support was added for SQL evaluation statements that can also be parameterized, as can be shown in the relation between **Q2** - **Q3**. Then, the grammar was extended to allow for variable declaration and for control-flow statements with any level of nesting. It is important to note that the conditional statements do not include intermediate **ELSE IF** statements yet; However, the plain **IF/ELSE** statements are equally powerful and lead to an identical (semantically) result. Expressions can be represented as N-ary operations, with the different precedence levels (e.g multiplication/division having higher precedence over addition/subtraction) being directly encoded in the grammar.

To quickly build this prototype from scratch, we utilized the libraries **JavaCC** [1] to generate the parser and **JTB** [3] to generate the AST and the matching Visitors.

3.1.2 Declaring Variables & Queries

Currently, the language supports only scalar values, booleans and table variables (where the information for the queries are stored). Every variable by default is initialized as immutable / constant following the logic of functional programming, making variable and query interpretation trivially parallelizable; This can be overridden by the user by adding the mutable type keyword **VAR** before the declaration:

```
// Immutable Query, result stored in TableVariable Q0
Q0 = SQL('SELECT AVG(col) FROM ... WHERE ... ');

// Immutable Scalar variable
y = 100.0;

// Immutable Scalar variable, parsed as a N-ary operation with different
  precedence levels
z = y + 10 - 5*5 + 4/2

// Mutable Scalar variable consisting of variables & literals
VAR x = y + z + 5
```

Listing 3.1: Examples of various declaration statements

If a Table Variable has been declared as mutable and it is defined by 2 or more query statements throughout the program, then all its previous data in its (main-memory resident) table gets deleted by the interpreter to allow for the new definition.

```
// Mutable Query, result stored in table0
VAR Q0 = SQL('SELECT AVG(col) FROM ... WHERE ... ');

// Immutable query using the result of Q0
Q1 = SQL('SELECT * FROM ... WHERE col2 = %q', Q0);

// Redefinition of (implicitly mutable) Q0, once Q1 has finished then table0
  is cleared and the new query gets issued
Q0 = SQL('SELECT SUM(col) FROM ... WHERE ... ');
```

Listing 3.2: Sample Mutable declarations

3.1.3 Control Flow Statements

A basic branch statement can evaluate arbitrarily complex conditions and contain any number of nested statements in the two branches:

```
Q0 = SQL('SELECT AVG(c.acctbal) FROM customer as c WHERE c.salary > 50000');
x = 0;

IF Q0 > 4495 THEN:
  x = 10;

  IF (x + 5 > 20)
    ...
  ELSE
    ...
ENDIF;
ELSE
  x = 20;
ENDIF;
```

Listing 3.3: Sample Branch statement

Similarly for the currently supported loop statements **WHILE** and **FOR**:

```
x = 0;
FOR i = 0; i <= 10; i = i + 1;
  x = i;
END;

WHILE x < 20;
  x = x + 1;
END;
```

Listing 3.4: Sample Loop statements

3.2 Front End: Parsing

In our system, a program dependency graph is used as the representation that illustrates the interplay of the queries and the constructs. The parsing mechanism - besides checking for syntax errors - is responsible for building the initial version of this graph, before handing it over to the back-end optimizer / rewriting component. For example, considering the sample snippet of Figure 1.1, it's PDG would be the following:

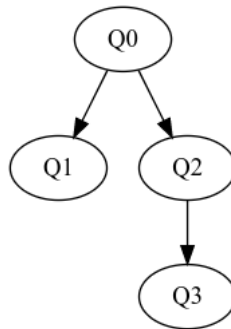


Figure 3.1: Execution flow of Figure 1.1 snippet

The flow of the Listing 1 snippet is now encoded in the directed acyclic graph (DAG) of Fig. 1, with **Q0** being at the root of the graph, as it must be fully executed before the correct branch is chosen. In addition, the inter-query dependency between **Q2** and **Q3** is depicted by these two queries being connected by an edge and residing in different levels, since **Q2** must finish before **Q3** is executed.

We have thus organized the (seemingly) scattered SQL statements in one packed representation and transformed the naive query-at-a-time execution into a promisingly scalable job scheduling problem.

It's pivotal to note that queries that are in the same level (e.g. **Q1** and **Q2**) are not dependent of each other, and can thus be visited (executed) by the scheduling algorithm in parallel. Consequently, the ordering returned from the topological sort if it ran on the dependency graph from Figure 1 would be:

$$\mathbf{Q0} \rightarrow \{\mathbf{Q1}, \mathbf{Q2}\} \rightarrow \mathbf{Q3}$$

3.3 Middle End: Speculative Execution

3.3.1 Relaxing Control Flow Dependencies

After the foundation of the code-aware engine has been laid out, we can introduce the optimizing component that performs rewriting passes on the initial dependency graph. Relaxing the control-flow dependencies would require a branch prediction mechanism that can - both quickly and fairly accurately - choose the most probable branch to execute in parallel with the evaluation of the condition e.g. query **Q0**.

In our work, AQP serves perfectly as the branch prediction mechanism, since it can compute a very close estimate of the queries that take part in conditional statements in a

fraction of the time that they would originally need, thus providing an approximate evaluation of the condition and consequently a prediction on which branch will be executed. It is also important to note that the trade-off between latency and accuracy can be tuned, by modifying the sampling size or process.

Let $Q0^*$ be the approximate version of $Q0$, with the properties that were described above. The rewriting pass will modify the dependency graph such that the condition query is now $Q0^*$, replacing $Q0$. In addition, the original condition evaluation is shifted to the branch level, which as described in Section 3.2, will result in the condition query being executed in parallel with the predicted branch. Thus, the control flow dependency has been relaxed, with the rewritten dependency graph shown in the following figure:

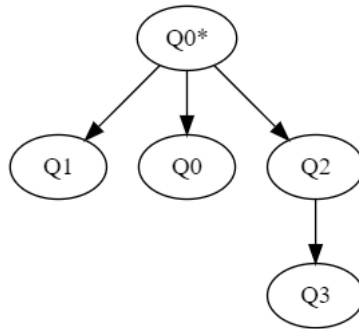


Figure 3.2: Relaxed execution flow of Figure 1.1 snippet

3.3.2 Repairing Mispredictions

Since $Q0^*$ produces only an approximate answer, it is easy to deduce that there is a considerable chance in which the prediction mechanism has chosen the incorrect branch to be scheduled. Let's assume that in the current example, $Q0^*$ finishes executing and the evaluated condition predicts the **IF** branch. Thus, our scheduling algorithm executes both the queries under the **IF** branch (e.g. $Q1$) and the initial conditional query $Q0$; In the near future, we expect that $Q0$ also finishes, together with some of the **IF** branch workload.

It is at this point that we must pause execution to compare our initial prediction with the result of the fully evaluated condition, which can lead to two potential outcomes:

- The prediction was correct; In this case, we have gained a considerable speedup by parallelizing the condition with the correct branch.
- The prediction was incorrect; In this case, we must repair the prediction by aborting the running **IF** branch workflow and initiating the **ELSE** branch.

Thus, we can summarize the speculative execution in the following steps:

1. Initially, the approximate condition is executed and evaluated.
2. Based on the result, a branch is selected, which is executed concurrently with the initial condition.

3. Once the original condition has been evaluated, the validation mechanism halts execution and decides whether it is required to repair the prediction or continue as is.

3.4 Back End: Interpreting & Scheduling Work

Once the parser has finished running, the AST of the imperative program has been generated. Utilizing the Visitor pattern, the AST nodes are extended such that they can accept a generic dispatch function call; Thus, one can define a custom "pass" over the syntax tree by implementing the behavior of a specialized visitor. In our system, these passes are the following:

Pass #1: The main focus of the first pass is to fill the Symbol Table with information regarding variable & query declarations and the program dependence graph (PDG), the structure that captures the data and control dependencies as previously explained. Lastly, this pass is also responsible for integrating the speculation framework, such as by generating the AQP equivalent queries Q_i^* for each query Q_i that is involved in a branch condition.

Pass #2: This pass uses the information collected from the 1st pass and begins interpreting each instruction; For example, at a variable declaration the right-hand side expression is dynamically evaluated and the resulting value is stored in the variable's Symbol Table entry. At a query declaration, the PDG is checked for dependencies with other queries/constructs, issuing the query if there are none. Once the query is finished, its node in the PDG is marked "disabled" to indicate that successor queries/constructs no longer have an active dependency and can also begin scheduling, as is explained in the Topological Sort algorithm.

4. EVALUATION

4.1 Setup

To build a prototype for the proposed framework and to be able to execute queries, an established open-source engine was needed that would serve as the foundation. For this reason, we chose Trino (formerly known as Presto) [5], a distributed query engine that is widely used in large-scale analytics.

Initially, our interpreter was integrated into Trino's front-end parsing mechanism, allowing it to accept and execute programs of the prototype language in the same manner that it would accept and execute standalone queries.

We also extended Trino's scheduler to account for the imperative programs; The enhanced scheduler accepts and parses a dependency graph instead of scheduling standalone query tasks. Since the speculative hints have already been integrated in the dependency graph, the original scheduler can still be used when scheduling and executing queries. Thus, the role of the enhanced scheduler is to visit the nodes of the graph in the appropriate order (as mentioned in Section 3) and to coordinate the speculative mechanism, while extracting the queries at each node and executing them through the original scheduling mechanism.

After executing each query, the intermediate results are materialized in temporary tables cached in memory. Then, whenever a variable is used in a control-flow statement condition or as a parameter in SQL statements, we can substitute its value by accessing the data of the appropriate table.

The optimizing component that performs the rewriting passes of the dependency graph lies between the parser and the scheduler.

4.2 Benchmarks

In this section, we evaluate the prototype of the proposed code-aware engine architecture that resolves control-flow dependencies using speculation. We then compare this framework with a query-at-a-time execution of the "correct" queries - that is, the queries that should be executed after having evaluated the IF/ELSE statements of the same imperative program - by a vanilla Trino configuration; this will mimic an engine executing the same workload without having resolved any control-flow dependency.

In the following benchmarks, we have used up to 3 levels of nested IF/ELSE statements. Each branch statement contains one TPC-H Q5 query in both the condition and at each branch, with a scale factor of SF=30 cached in memory. The Q5 was specifically chosen as it is a join-heavy query that doesn't scale properly; Thus, it will further drive the point that a solely data-parallelism execution can quickly reach diminishing returns, if the queries executed contain anti-parallel patterns and no strategies are employed to increase query parallelism.

Both engines are run on a 1-node cluster, equipped with 370GB of RAM and 2 × Intel Xeons 5118 containing 12 cores each.

In both benchmarks, the execution time in relation to the nested IF/ELSE levels is depicted

in blue bars for our speculative framework and in orange bars for the query-at-a-time execution (non-speculative).

Figure 4.1 (below) shows the end-to-end execution time speedup gain for solely correct predictions that vary in nested branch levels, ranging from $\times 1.25$ up to $\times 1.5$.

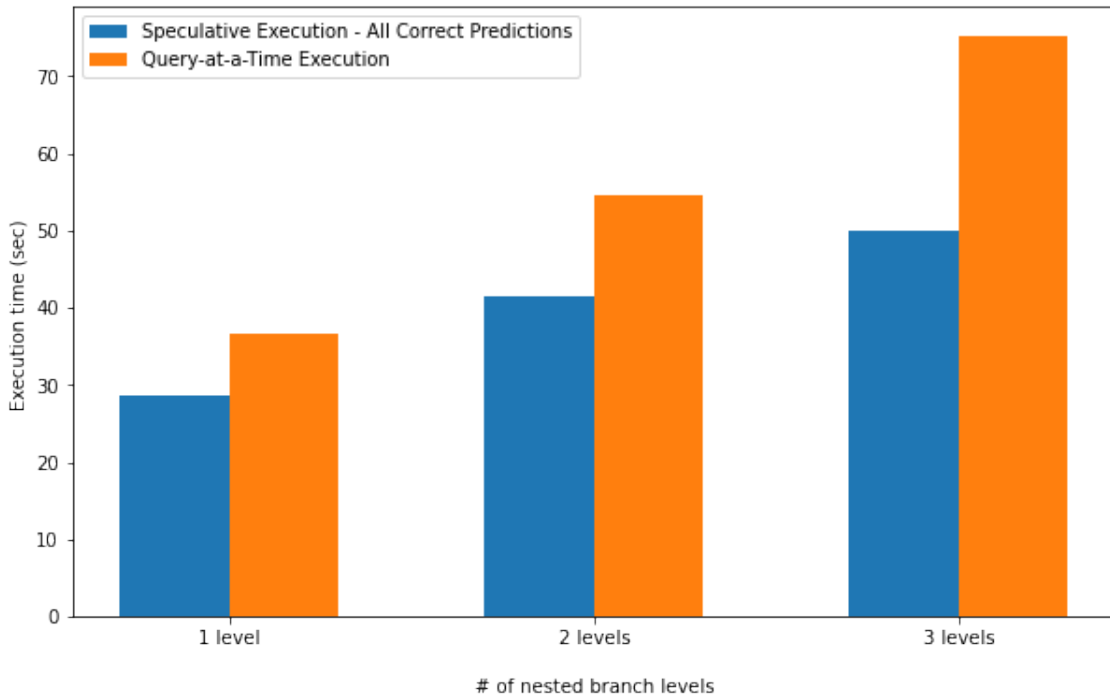


Figure 4.1: Execution time of all correct predictions vs Non-Speculative version

Figure 4.2 shows the repair overhead accounting for an increase of $1.1\times$ (avg) in end-to-end execution time, for mispredictions in every level.

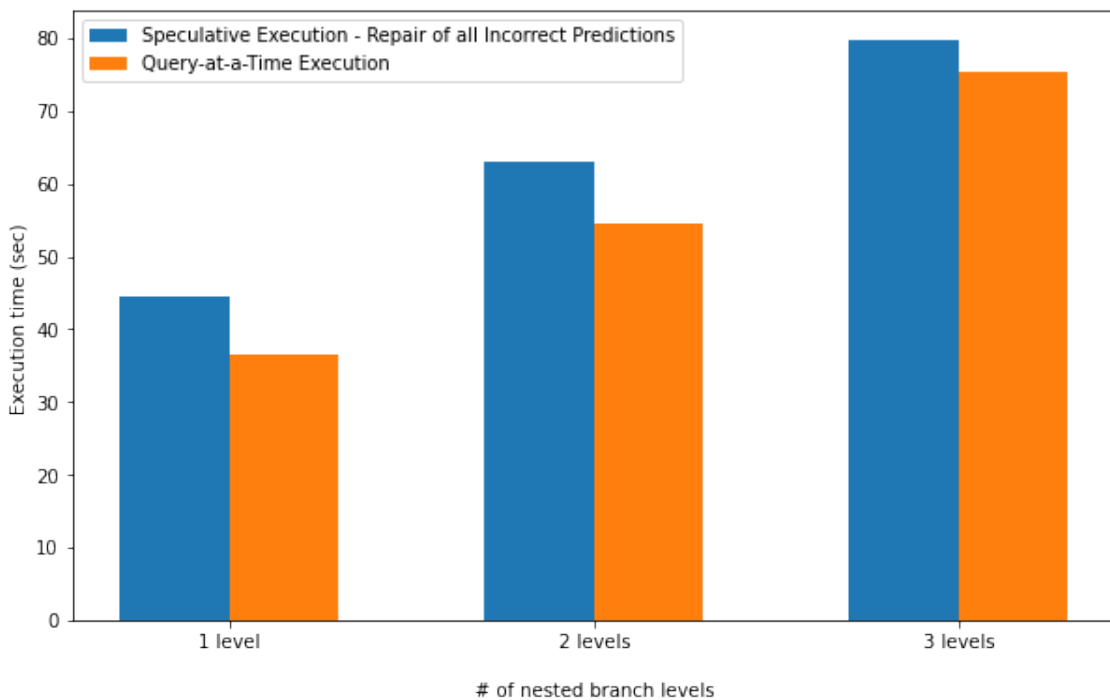


Figure 4.2: Execution time of all incorrect predictions vs Non-Speculative version

5. CONCLUSIONS AND FUTURE WORK

In this thesis, we first propose a new architecture for OLAP engines that execute mixed imperative-SQL analytic workflows, in order to increase synergy between the components. This allows us to implement a speculative mechanism that relaxes control-flow dependencies, thus increasing parallelization opportunities.

We demonstrate that, in an optimistic scenario, this can result in an average acceleration of a factor of 1.4× in nested conditional statements. In comparison to branch mispredictions, a small overhead is induced, increasing the end-to-end execution time by a factor of 1.1×. Thus, even though a greater suite of experiments is needed to accurately provide an average speedup vs repair cost trade-off, these initial results indicate that relaxing control-flow dependencies using a speculative framework is a promising feature, as it has been already proved in modern hardware.

In the future, we hope to extend the interpreter by bridging the established optimizations in compilers such as common subexpression elimination, constant folding, loop unrolling etc with the equivalent optimizations in databases, such as data sharing. These are not only critical in increasing the interpretation efficiency, but can also unlock further optimization opportunities. For example, by unrolling a loop that contains queries in its body, we not only harness the potential to run loop iterations in parallel, but can also benefit e.g from a data sharing framework that is capable of recognizing the reusable data resources and caching them, thus reducing the overall memory footprint and removing the cost of materializing multiple tables with overlapping data across the unrolled queries.

ABBREVIATIONS - ACRONYMS

AST	Abstract Syntax Tree
PDG	Program Depedence Graph
DAG	Directed Acyclic Graph
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing

BIBLIOGRAPHY

- [1] Javacc: The most popular parser generator for use with java applications.
- [2] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Ion Stoica, and Samuel R Madden. Blinkdb: queries with bounded errors and bounded response times on very large data. In *In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 29-42., 2013.
- [3] UCLA Compilers Group. Jtb: Java tree builder.
- [4] Sioulas Panagiotis, Sanca Viktor, Mytilinis Ioannis, and Anastasia Ailamaki. Accelerating complex analytics using speculation. In *11th Annual Conference on Innovative Data Systems Research (CIDR '21), January 10-13, 2021, Chaminade, USA.*, 2021.
- [5] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezhir Yigitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: Sql on everything. In *Facebook, Inc.*, 2018.
- [6] Wikipedia. Optimistic concurrency control.