Title
Bridging the Gap Between Application Logic and Auto-optimization in Modern Data Analytics

Author
Ramjit, Alana Morgan

Publication Date
2021

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Bridging the Gap Between Application Logic and Auto-optimization in Modern Data Analytics

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Alana Morgan Ramjit

2021

ABSTRACT OF THE DISSERTATION

Bridging the Gap Between Application Logic and Auto-optimization in Modern Data Analytics

by

Alana Morgan Ramjit
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2021
Professor Ravi Arun Netravali, Co-Chair
Professor Todd D. Millstein, Co-Chair

Recent decades have seen an explosion in the diversity and scale of data analytics tasks. While data analysis of the late 20th century was characterized by the dominance of relational databases and highly structured querying languages, demand for less structured and more complex tasks has resulted in new data analytics frameworks that break with the norms of historical systems.

This growth has come at the cost of breaking with the assumptions that guided automated optimization in historical systems. Automated optimizations analyze the input program of a system and extract insights that allow the system to better execute a given task with little to no human effort. Absent these features, analysts must manually tune data processing frameworks to achieve reasonable performance, a delicate and time-consuming endeavor.

In this thesis, I argue that automated optimization techniques, such as caching and physical design, that have long been deployed in relational frameworks remain both relevant and necessary to achieving sustainable performance in modern systems. Via two projects, each targeting a different class of analysis tasks, I identify new techniques to bridge the gap between modern data analytics frameworks and established automated optimization techniques.

The dissertation of Alana Morgan Ramjit is approved.

Eugene Wu

George Varghese

Harry Guoqing Xu

Todd D. Millstein, Committee Co-Chair

Ravi Arun Netravali, Committee Co-Chair

University of California, Los Angeles

2021

For the girls, the goths, and the gays.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I cannot express enough gratitude to my brilliant and kind advisor, Ravi Netravali, for his unwavering support throughout all of my indecisiveness and struggles over the past few years. It is already clear that he is one of the rare advisors who cares as deeply about his students as he does about the quality of his research. Lesser advisors would have given up entirely during my third proposed research direction change, but not Ravi; I was the first advisee he had, probably the most annoying, hopefully the worst he ever has to contend with. I am likewise deeply indebted to my co-chair, Todd Millstein. He may not have been my advisor on paper, but he was certainly one in spirit. Thank you for always offering guidance and wisdom and most especially during the rockiest part of my graduate career.

I would also like to my other committee members, George Varghese and Harry Xu for their service on my committee and their much welcomed presence in the UCLA computer science department. Eugene Wu at Columbia provided indispensable technical advice, patience, and support both in my research and career; his ideas and feedback are critical for all of the projects described in this thesis. I am honored that my committee was composed entirely of the most caring and intelligent faculty members I could imagine.

I owe additional thanks to collaborators and reviewers: for my work in Chapter 3, Matteo Interlandi for his feedback and guidance during the project, as well as comments, ideas, and additional feedback from Tyson Condie, Miryung Kim, and Databricks. In Chapter 4, Subrata Mitra's input was crucial for identifying and continuously shaping key ideas.

Finally, the last six years would have been impossible without the support of friends, family, and community. It is one of the most delightful surprises and blessings of my life that no matter where I turn, I seem to find wonderfully kind, caring, intelligent, and supportive people: my friends in LA, in NYC, my neighbors, my roommate, Joseph Brown in Student Affairs (a real one), my cat Baby, my Runescape clan–all of you have offered advice, commiseration, and solutions depending on what I needed most. Thank you.

VITA

2015	B.A. in Computer Science Columbia University New York, NY
2015-2020	Graduate Student Researcher Computer Science Department University of California, Los Angeles
2015-2017	Teaching Assistant CS 31: Introduction to Computer Science and CS 97: Principles of Programming Computer Science Department University of California, Los Angeles
2016	Research Intern Mozilla, Inc San Francisco, CA
2017-2018	Network Performance Intern Verizon Digital Media Los Angeles, CA

PUBLICATIONS

Lana Ramjit, Matteo Interlandi, Eugene Wu, Ravi Netravali. *Acorn: Aggressive Result Caching in Distributed Data Processing Frameworks*, SoCC 2019. Nov 2019.

Lana Ramjit, Zhaoning Kong, Eugene Wu, Ravi Netravali. *Physical Visualization Design (demo)*, SIGMOD 2020. July 2020.

CHAPTER 1

Introduction

Data analytics broadly encompasses any task that seeks to manipulate data, a process that allows human analysts to explain, understand, or explore diverse types of information. For decades, data analytics was nearly synonymous in computing with the relational querying model popularized in the 1970s. One of the driving forces behind the early dominance of the relational model was its explicit accommodation of automated optimization, which developers quickly realized would be essential for *efficient* data analytics in realistic environments [Cod02]. Automated optimizations analyze the input program, extracting insights that allow the framework to modify its execution to optimize for performance, resource consumption, or some other desirable goal–all with little to no programming effort on the part of the human data analyst.

The relational model was designed with the intent of making data processing frameworks amenable to automated optimization. It introduced the notion of data independence, a principle that separated the description of data processing tasks from any knowledge of how data was physically stored. Instead, data analysts could use the abstractions provided by the relational model to reason logically about the end result of their programs (*what to do?*) while delegating the complexity of finding an efficient execution plan (*how to do it?*) to the underlying system. This was facilitated by declarative languages which implemented a mathematically precise set of operations, establishing a set of coherent guarantees between the program and execution system: regardless of how the framework actually executed each operator, it only needed to fulfill the guarantees enforced by each operator.

The relational model emerged as a standard in part because of how difficult finding an *effi-*

cient execution plan is in practice. Seminal efforts towards establishing automated optimization highlight the inherent complexity of the task: an efficient execution framework needs to generate a set of potential execution strategies, accurately estimate their cost, and select a reasonably well-performing plan, all without introducing self-defeating overhead [DDD04, CN97, CN07]. Because execution strategies and cost estimation both vary depending on not just the individual task and capabilities of the underlying system, but also the available resources and data size, this work needs to be repeated when any of these variables change significantly. Absent automated optimization, it falls to analysts to decide under each of these circumstances how the data processing framework should best optimize the execution of each input program and write the corresponding code, a pain-staking, error-prone, and time-consuming endeavor.

In recent years, access to cheap data storage and increased computing power has rapidly expanded, enabling data analysts to capture even larger amounts of data and process it in increasingly complex and diverse ways. This trend has ushered in a corresponding growth in the ecosystem of automated data analytics frameworks that facilitate different types of tasks, including those that are not supported by traditional relational frameworks. Unfortunately, many of these frameworks are no longer amenable to the automated optimization techniques developed in relational databases for two principle reasons.

First, to support the growing complexity of data analytic tasks, many of these frameworks are intended to execute operations that are not supported by declarative languages. Declarative languages define a set of mathematical operations that establish a crucial contract between the intended logic of the program and execution strategy of the system. However, if an analyst needs to perform an operation that is either not defined or not easily expressed by combining these predefined operations, then they must use an auxiliary language, hampering the ability of the execution system to find efficient execution plans.

Second, automated optimizations are closely tied to the frameworks that implement them. A fundamental task of automated optimization is to generate a pool of execution strategies and efficiently search for the a relatively efficient approach; naturally the pool of execution strategies

depends directly on the capabilities of the underlying system. As data analytics has shifted to include new modalities of data processing, data processing frameworks likewise must shift to explore new potential execution strategies.

In this thesis, I argue that existing automated optimization techniques can bridge the gap that has emerged between application logic and modern data processing frameworks in one of two ways. They can analyze program and execution context and reconfigure themselves into existing paradigms, by, e.g mapping application logic into existing declarative operators and re-organizing their pipelines to mirror their predecessors. Alternatively, they can augment the language capabilities and pool of execution strategies to better reflect modern demands, defining new declarative operators and corresponding execution strategies to implement those operators. Throughout the following two projects, we explore both of these methods, porting automated optimization into frameworks representing two significant trends in modern data processing: distributed execution and interactive visualizations.

Acorn: Aggressive Caching in Distributed Data Processing Frameworks identifies two key hurdles common in modern distributed data processing systems that prevent these frameworks from taking full advantage of aggressive caching techniques. Distributed data processing systems such as Spark, Pig, and Presto are designed to scale the processing of very large datasets across multiple computation centers. Programs written in these systems use a mix of the declarative, relational operators from traditional databases, along with user-defined functions written in general purpose programming languages. As we show in Chapter 3, both the input languages and execution pipeline introduce obstacles to effective caching. `Acorn` demonstrates how simple pipeline reorganization combined with powerful translation techniques can circumvent enable the automated and aggressive caching, significantly improving the performance of several common classes of data analytics tasks.

Jade: Physical Visualization Design is a middleware system that manages the *physical data layout* of interactive visualization applications. Interactive visualizations facilitates data analysis by providing a visual representation of data that users can then inspect for patterns, or modify

via interactions that sort, search, filter, and otherwise manipulate the represented data. These applications have specific performance requirements in order to remain responsive on a timescale corresponding to human cognition, and developers of interactive visualization applications must analyze the intended application and build custom execution architectures to meet these performance requirements. `Jade`, described in Chapter 4, provides an automated solution to this, adapting classical physical design techniques used in relational databases to the unique, domain-specific requirements of interactive visualizations.

CHAPTER 2

Background

Both projects described in the following chapters make frequent discussion of the relational algebra, its most commonly used implementation, Structured Query Language (SQL), and the databases that execute them. Section 2.1 begins with a basic introduction of the relational algebra and SQL that can safely be skipped by those with a passing familiarity. Section 2.2 follows this with a description of execution pipelines and relevant automated optimization techniques found in relational database management systems, given as grounding for the techniques described in following chapters.

2.1 The relational model

Prior to the adoption of relational data, information retrieval was graph based. Navigational databases linked units of data together with pointers, much like modern file structures. Users who wished to query data would describe how to navigate to the items they wished to retrieve by following pointers through the data storage unit. The primary drawback of this querying paradigm was that information retrieval code was closely tied to file storage; if pointers were moved or broken, then the application logic became obsolete.

The relational model, introduced by Codd in "A Relational Model of Data for Large Shared Data Banks" pioneered the idea of data independence: by providing an abstraction for the organization of data that was independent of the underlying file system, application logic separated information retrieval from information storage. In this abstraction, data is modeled as a set of *re-*

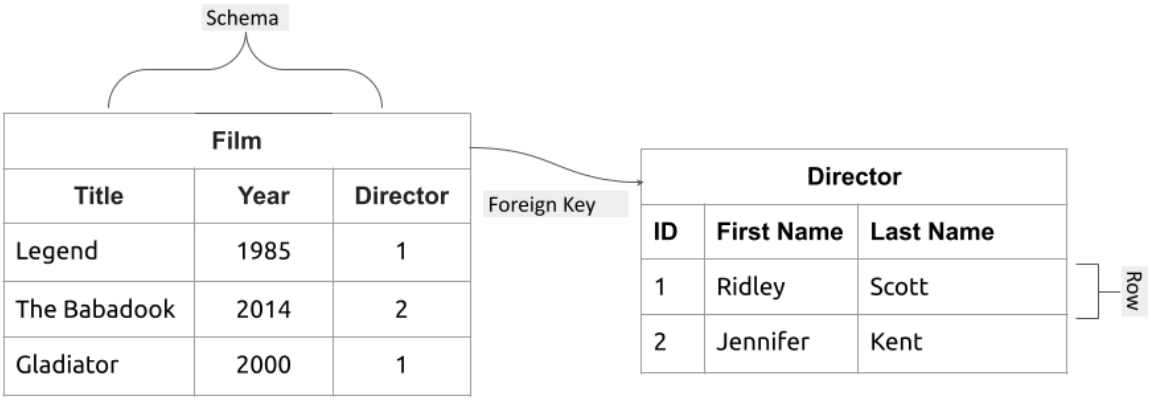


Figure 2.1: A simple example of a relational schema, in which the relation `Film` references records in the table `Director`.

lations, also called *tables*. Each relation has a defined *schema* that bestows a title and data type to each column in the table. Consequently, each row (or *record*) in the table has a well-defined, uniform structure. Each row may optionally have a unique ID, a *primary key*. A table can describe relationships between individual records in another table by using a `foreign` key to indicate a unique record in another table; an example of such a set of schemas is shown in Figure 2.1. The advantage of the relational model, as Codd argued, was that it "provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other." The proposed language has mathematical foundations known as the relational algebra, which defines allows operations on one or more relation. Each operation in the relational algebra is a simple primitive that defines the semantics for performing a single operation. A query in the relational algebra composes these primitives to specify a series of intended transformations. What follows is an overview of the primitives most relevant to this thesis.

Select-Project-Join Taken together, `Select-Project-Join` form the operations in the most basic set of relational queries, known as `SPJ` queries.

Select The `select` operator, σ has as an argument a predicate that evaluates to true or false. Given a relation R, it will produce a new relation with the same schema but consisting of the subset of records for which the predicate evaluated to true. Using the set of relations in Figure 2.1 as a basis, the selection expression $\sigma_{Year>2000}(Film)$ will produce a single row:

Title	**Year**	**Director**
The Babadook	2014	2

Project The `project` operator, π produce a new relation with only the subset of columns specified by its argument. The projection expression $\pi_{Title,Year}(Film)$ will, for example, produce the new relation:

Title	**Year**
Legend	1985
The Babadook	2014
Gladiator	2000

Join Finally, the `join` $\bowtie_\theta$ operator takes as input two tables and produces as output the combination of both tables where the predicate θ is true. For example, the operation `Film` $\bowtie_{Director=ID}$ `Director` would produce the following table:

Title	**Year**	**Director**	**ID**	**FirstName**	**LastName**
Legend	1985	1	1	Ridley	Scott
The Babadook	2014	2	2	Jennifer	Kent
Gladiator	2000	1	1	Ridley	Scott

There are many common extensions to the `join` operator omitted from this discussion, as it suffices to be familiar with its most basic form.

Set operations For relations that have identical schemas, *set operations* are supported in the relational algebra. The union of two relations, R$\cup$S is a relation containing all non-duplicated records

in either R or S. The intersection of two relations, R ∩ S is the relation consisting of all records that appear in both R and S. The difference operation, R S produces a relation with all items in R that are not in S.

Grouping and Aggregation Grouping and aggregation were added as an extension to the original relational algebra. Combined with the `Select-Project-Join` primitives, these operations form a class of queries commonly referred to as SPJGA queries.

An aggregation expression $\mathcal{G}$, or equivalently λ, specifies one or more functions used to combine multiple records. Common aggregation functions include `sum`, `average`, `min`, and `max`. For example, the expression $\lambda_{count(title)}$(`Film`) counts the number of (non-distinct) title entries in the film table, producing the following table:

count(Title)
3

Aggregation expression may also combine records only if they share a value in a certain column. For example, the expression ${}_{director}\lambda_{count(title)}(Film)$ groups records that share the same `director` value, and aggregates each group by counting the number of (non-distinct) `title` entries in each group.

Director	**count(Title)**
1	2
2	1

UDFs and other operators Many other operators have been proposed as extensions to the core SPJGA operations. Different implementations and frameworks may provide support for all or some subset of these additional operations. However, for desired operations not supported directly by a database querying language, many systems provide support for *user-defined functions (UDFs)*. UDFs are functions written by a developer that take as input one or more columns as arguments,

perform some operation on each row on the column, with the output value constituting a corresponding row entry in a resulting column. For example, a user may define the following UDF that squares an integer input value:

```
def squaredUDF(col: Integer){
   return col * col;
}
```

Then the expression $\pi_{squaredUDF(Director)}(Film)$ would produce the relation

squaredUDF(Director)
1
4
1

UDFs are useful for expressing non-standard operations; however, since UDFs are custom functions, databases typically execute them as *blackboxes*, e.g. with making no assumptions about the expected output or execution of the function. Thus, their use is discouraged since they typically lead to poor performance.

2.2 Automated optimization in relational engines

In the relational model, a query is a combination of operations in the relational algebra. Several declarative languages, most notable Standard Query Language (SQL) implement and define a syntax for writing queries over the relational algebra with various extensions. Relational database management systems (RDBMS) are the data processing engines responsible for taking an expression in the relational algebra and (efficiently) evaluating it against a physical data storage. There are many ways to evaluate a given query but not all do so efficiently; the process of enumerating and selecting an evaluation plan is known as *optimization*. In this section, we give a brief overview of the optimization process, broken into two stages: query planning and physical design. In the

former, the engine considers how data is currently physically stored internally and decides on a set of operations to execute a given query. In the latter, the engine, given some information about the expected workloads, makes decisions about how to organize the data it stores in order to best generate efficient query plans for that workload.

2.2.1 Query Planning

While query planners are generally quite complex, most query planners used in RDBMS share a similar pipeline. In this pipeline, there are three phases that are relevant to our discussion: parsing, optimization, and physical planning. In the first phase, queries written in a declarative language are **parsed** into a logical plan, an internal representation that transforms the set of operations described in the query into a tree-like structure.

Next, during optimization, each relational expression in the logical tree undergoes a series of transformations that result in a standardized format and ordering. Optimizations that happen in this stage are known as algebraic optimizations or rewrites. Rewrites are guaranteed to be semantically equivalent (i.e. always produces the same result when evaluated). One example of an algebraic rewrite is constant propagation, in which a constant expression (e.g. 1+1) is replaced with the constant 2.

Another example is re-ordering expressions within the logical tree. For example, a rewrite known as "predicate pushdown" results in a query plan where σ operations happen earlier rather than later. This frequently results in a more efficient plan, because it removes unnecessary records as early as possible, requiring the engine to scan fewer records when evaluating later expressions. Typically, these optimizations do not depend on the physical layout of the data.

Finally, the optimized logical tree is **planned** into a physical plan, a process in which the engine decides how and in what order to physically access the data and compute each expression. The most basic physical plan is to retrieve a table and scan it linearly applying each expression in their original order to each record. In practice, this plan is almost never used in favor of more

efficient plans found during optimization, but physical planning frequently depends on information (*metadata*) about the physical data in the RDBMS: e.g. which access paths are available? How big is each table? How frequently does a specific value occur?

For example, the operation $\sigma_{Year>2000}$ can be executed by scanning the `Year` column of a table and accumulating each row that evaluates to true. However, this requires scanning each individual row in the table. A more efficient plan might take advantage of an additional *access path* available for the table. RDBMS have the ability to create secondary indexes, which sorts the records by the values in each column. For example, an index on the `Year` column would ensure all rows in the table are sorted by year. Then, the expression $\sigma_{Year>2000}$ only needs to find the first row with the year 2000 and can then retrieve all remaining records. Which access paths are available depends on which indexes (and other supporting structures) are currently created in the database, and the process of deciding which access paths to make available is known as physical design.

2.2.2 Physical design

In addition to storing raw tables, RDBMS often also store duplicates of the data in secondary structures. The decision of which structures to create in support query planning is known as physical design. As described in the previous section, the RDBMS engine's freedom to generate efficient plans is highly dependent on which secondary structures have been created, thus physical design is crucial in maintaining reasonable execution times on many query workloads.

Secondary structures can loosely be divided into two categories. First, some data structures replicate the entirety of the original input data but provide new access paths, such as indexes which sort tables according to a particular value. Second, cached data stores the results of previously executed queries, and typically only replicate a subset of the original data; this includes structures such as materialized views or precomputed aggregates. Caching is best used for shared work across multiple queries against data that are not expected to change often. For example, if two tables are frequently joined together in the same manner across multiple queries, then caching the result of

this join operation will help execute all such queries faster.

CHAPTER 3

Aggressive Caching in Distributed Data Processing

3.1 Overview

Recent years have witnessed significant efforts to improve the speed with which large-scale data processing frameworks like Apache Hadoop [Apa17] and Spark [ZCD] execute queries [KA10, ZKJ08,NBD12,ORR15]. A common technique used to accelerate data processing tasks is result (or view) caching [GMb, MB, TS, ZLF07]. With this optimization, results from prior query executions are used to reduce the on-demand work needed to execute new queries.

Result caching has been successfully employed by traditional databases and early data processing frameworks [JMH16, Hal01, CY12, EA12, NPM10, CCH16, GRT10, JKR18, KFM17]. Benefits have been particularly pronounced for the iterative workloads common to machine learning algorithms, and the incrementally constructed queries in graph processing and interactive data exploration sessions [AXL15]. However, several major trends in recent data processing frameworks (e.g., Spark SQL [AXL15]) complicate the use of result caching.

First, many distributed data processing frameworks generate increasingly large query plans which are both expensive to execute *and* expensive to optimize [ZCD12]. The reason is that, unlike databases which perform data updates in-place, modern analytics frameworks operate on immutable data [ORS, BBE15, AXL15]. This model treats data as read-only, and updates or queries that data by maintaining a lineage of transformations whose intermediate results may be materialized. Although this simplifies debugging and failure recovery [ZCD12], transformation histories (and query plans) can grow to immense sizes, particularly for iterative and incremental workloads.

Aggressive result caching is a natural way to shrink query optimization overheads. However, while there has been much work on deciding what results to cache [AX, RSS00, RRS00, FMC09], modern frameworks still struggle with determining how to make the best use of cached results. Frameworks such as Spark SQL elect to apply exact-match caching, rather than more powerful techniques like predicate analysis that can also identify partial query equivalence matches (i.e., where the results for one query entirely or partially subsume the results for another query) [GL01]. The reason is that it is challenging to determine where in the query optimization pipeline (Figure 3.1) predicate analysis can be efficiently performed. Performing predicate analysis before query optimization can reduce query plan sizes and optimization costs, but requires operating on *analyzed query plans* (logical query plans that have not gone through the optimizer) which obfuscate caching opportunities since predicate pushdown has yet to be performed. Predicate analysis after query optimization can benefit from optimized (canonicalized) query plans, but must fully incur expensive optimization overheads.

Second, modern analytics frameworks increasingly make it easy for developers to interleave declarative querying with user-defined functions (UDFs) expressed in general-purpose programming languages (e.g., Java, Scala). For instance, 74% of DataBricks' [Dat] client-facing clusters run workloads that contain UDFs, with UDF execution accounting for 34% of median cluster execution time.[1] This trend will likely grow, as in-language integration of data flow engines increases, making UDF-heavy analytics programs easier to write [AXL15, FBE09, McK11].

Unfortunately, query optimizers treat UDFs as black boxes, and must thus resort to exact-match caching. Recent work such as Froid [RPE17] shows how UDFs written in special SQL procedural languages (i.e., T-SQL) can be translated into native SQL operator plans. However, Froid provides limited support for UDFs expressed in general purpose languages, as Froid may not preserve types and cannot support language constructs such as generics, reflection, and virtual function invocations.

[1] Databricks only provided these statistics, not raw workloads.

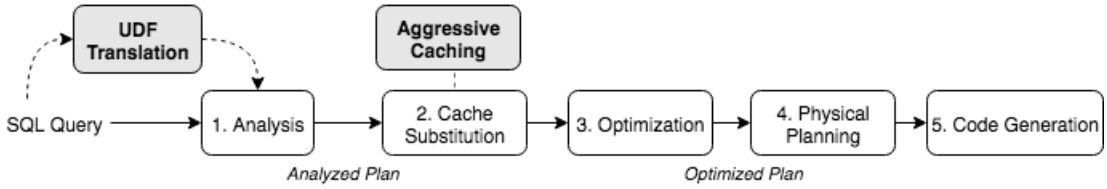


Figure 3.1: Spark SQL's query planning pipeline. Grey boxes and dotted lines indicate `Acorn`'s new components. Each stage generates or modifies a query plan and passes it down the pipeline; query plans relevant to `Acorn` are labeled.

This paper addresses the use of result caching in large-scale data analytics frameworks through the combination of judicious adaptation of existing techniques such as predicate and program analysis, and novel UDF analysis for general-purpose languages. Our goal is to enable aggressive result caching without 1) burdening developers to provide hints or rewrite queries, 2) incurring unnecessary query optimization overheads, or 3) sacrificing the expressiveness of UDFs. We integrate our ideas in Spark SQL, but the problems we tackle and our solutions broadly apply to large-scale data processing frameworks (§3.7.3). We make three main contributions.

Our first contribution adapts the extensive caching and predicate equivalence concepts from the database community [GL01, LSH14] to distributed query processing frameworks. Rather than ineffectively performing predicate analysis on analyzed query plans or operating on optimized query plans that have already incurred considerable optimization costs, our key insight is to perform a cheap *partial optimization* pass that applies only the handful of optimizations (e.g., constant propagation, predicate pushdown) that affect predicate analysis. In this way, queries only run through the full optimizer *after* aggressive caching decisions are applied. Our predicate analysis identifies total and partial subsumption relationships between analyzed plans and cached results.

Our second contribution translates UDFs written in a general-purpose language into equivalent functions expressed solely with native query operators and API calls. This opens up UDFs to the query optimizer, including the result caching mechanism, and enables the co-optimization of UDFs and relational queries. To do this, we lower UDF Java bytecode into a type-preserving intermediate representation, and use symbolic execution to quickly generate an equivalent query plan.

Translation is completely transparent to developers (unlike VooDoo [PMZ], Weld [PTN18]), and can support the advanced language features described above that Froid [RPE17] cannot. Although our approach supports almost all Java and Scala features, it is best-effort and is mainly limited by the target language (e.g., Spark) (§3.6.3).

Our third contribution is **`Acorn`**, an implementation of the aforementioned result caching optimizations in the latest version of Spark SQL (v2.4). We evaluated `Acorn` on two benchmark workloads (TPC-DS and TPC-H [tpc]) with datasets sized between 1-100 GB, as well as on multiple real-world Spark workloads. Experiments show that `Acorn` provides speedups of 2× and 5× over Spark SQL for benchmark workloads with and without UDFs, respectively, while imposing negligible overheads and no changes to the workloads. Benefits were 1.4×–3.2× for real graph processing workloads. Further, other than 3 UDFs that are not expressible with Spark's native API, `Acorn` was able to translate all UDFs in our workloads, many of which Froid cannot.

3.2 Background

Using Spark SQL [AXL15] as an example, we discuss how distributed data processing frameworks plan and execute queries, and how these design decisions affect result caching and their programming environments.

Query Planning: Spark SQL's query planner, Catalyst, advances queries through five phases to translate a logical query plan into a physical one where operators have been moved, replaced, or combined based on optimizations and disambiguation rules (Figure 3.1). In stage 1, the analyzer resolves column names to a table or dataset, validating any column or table references in the query, and outputting an *analyzed query plan*. If a query has previously been marked for caching (described below), stage 2 retrieves the cached data by exactly matching the stage 1 plan to the cache index. Stage 3 applies a set of rules to the (potentially modified) analyzed query plan, restructuring and rewriting the query plan for efficiency, and outputting an *optimized query plan*. Stages 4-5 choose physical operators for the query, such as a particular join strategy, and generate

code for executing those operators. Many rule-based query planners [Cha98, GM91, Apa17] share these steps, but Catalyst slightly differs in that stages 2-5 are lazily evaluated for efficient in-memory execution [ZCD12].

Caching: Spark SQL uses immutable datasets and maintains a lineage graph (akin to a query plan) associated with each materialized dataset. To minimize optimization times (§3.1), Spark SQL performs caching prior to query optimization (stage 2 versus stage 3), allowing cache hits to skip the optimizer. Caching in Spark SQL is primarily user-driven, where users explicitly mark intermediate results as cacheable.[2] When a user requests that a query or dataset is cached, the Stage 2 cache manager creates an index using the current analyzed plan (from Stage 1), and reserves space for the to-be-computed dataset. As of the latest version of Spark SQL (v2.4), the cache manager uses a hash-based canonicalizer to match analyzed plans during cache substitution. Thus, cache hits only arise when the cache manager finds an *exact match* for the corresponding analyzed plan. This differs from traditional databases, where cache substitution happens after the optimization step, and is thus performed on optimized query plans that facilitate the detection of caching opportunities through in-exact matches [GL01, IR95, ZND01, CR00].

Programming Environment: Spark SQL maintains a DataFrame API that enables seamless integration of procedural and declarative tasks. Although raw SQL strings are accepted, developers can write valid SQL queries by chaining procedural API calls that mirror SQL clauses. For example, a projection clause typically found in the `SELECT` clause of a SQL query can be written using the `select()` function in the DataFrame API. The DataFrame API also provides support for data processing tasks such as those in MapReduce systems [AXL15].

Spark SQL also allows users to write *user-defined functions* (UDFs) in general-purpose programming languages (e.g., Scala, Java, Python). UDFs can be registered with the query engine (as they are in traditional databases), or they may manifest as lambdas and anonymous closure functions. Certain UDFs may be used as operators, being mixed into a chain of native API calls.

[2]Recent implementations [Dat] support automatic caching, where intermediate results are optimistically cached without user instruction.

However, anonymous UDFs can only be passed as arguments to existing operators. Importantly, both forms of UDFs are treated as black boxes by the query engine, which is unaware of how a UDF will access or manipulate data. Registered UDFs preserve their user-given name across appearances, while anonymous UDFs get a unique name each time. Thus, the former can benefit from exact-match caching, while the latter cannot (the changed name results in a changed analyzed plan).

3.3 Related Work

Multi-query Optimization: Materialized views [GL01,IR95,ZND01] and their maintenance [ZLE07, MRS01, GMa] has received much attention in databases. We draw on many of these principles to find containment relationships. However, our focus is on efficiently applying these techniques to the new domain of data analytics frameworks with lazy query planning and UDFs. Further, view maintenance does not apply in this environment since source relations are immutable (unlike with databases). Other systems [NPM10] find work-sharing opportunities within intermediate results for queries executed simultaneously. Our approach is complementary as we target result caching for queries that are handled separately.

Result Caching in Data Processing Systems: ReStore [EA12] employs result caching and incremental computation in MapReduce-like systems but uses graph-based searches to compare physical operators. Unlike `Acorn`, ReStore requires optimized query plans and also ignore UDFs. PigReuse [CCH16] provides an alternative using predicate analysis on analyzed query plans. However, PigReuse's methods are specifically designed for restricted variants of PigLatin; we focus on general purpose languages with many more operators. CloudViews [JKR18] finds useful subexpressions for caching in shared cloud jobs. This, and similar related work which adds the caching annotations assumed by Acorn, is fully complementary to our goal, although unwrapping UDFs would strengthen such annotations.

Domain Specific Languages for Data Processing: Several prior approaches provide new, heav-

ily optimized languages for data analytics environments [PMZ, PTN18, Mou18]. These techniques provide expressive (but performant) languages that are alternatives for efficiently writing and evaluating UDFs. However, using these languages requires manual rewriting of workloads. Instead, `Acorn` transparently accelerates unmodified workloads.

Optimizing UDFs: Several systems parse UDFs and extract information to aid the optimization of program execution [CGD15b,CGD15a]. These approaches are orthogonal to `Acorn`, which can be modified to extract similar optimization properties during UDF translation. This, however, would require changes to Spark SQL's query optimizer; in contrast, `Acorn`'s components are transparent to downstream pipeline components.

Bytecode analysis can extract key properties from UDFs that are strong enough to enable reordering [HPS12, RHH15]. However, the derived annotations are not strong enough to detect subsumption relationships for aggressive caching [DN14, RLG17]. Other approaches have detected subsumption relationships with UDFs [LSH14], but require manual UDF annotation and only work with registered UDFs (limiting benefits for Spark SQL where anonymous UDFs are common).

Perhaps closest to our approach is Froid [RPE17] which inlines UDFs into SQL queries. Froid translates each statement in isolation, cannot ensure type-safety with generics, and requires a separate, customized mapping class for each imperative construct. Instead, `Acorn` uses symbolic execution (traditionally used for model checking and constructing logic formulas) to dynamically connect sequences of statements. This is critical, since Java bytecode frequently erases generic types, which are extensively used to declare lists and SQL relations.

For example, a UDF might create and fill a `Row<T>`. Froid's parse-and-map strategy does not allow user-defined types, let alone generics: because of bytecode type erasure, Froid would create a table of Objects for use in subsequent statements in the UDF. This introduces exceptions if the Row is later cast back to the original type. Instead, symbolic execution lets `Acorn` treat the variable as `Row<Object>` until a cast is performed or a specific subtype is found, update the type, and then propagate this type information to subsequent statements. Handling these subtle details lets `Acorn` support UDFs with generics, reflection, and virtual function invocations, which

Froid cannot. Thus, while Froid can effectively handle the constrained T-SQL language, it is intractable (and unsafe) for general purpose languages.

Program Equivalence and Synthesis: Certain systems use equivalence detection techniques [SSS16, CWC17] or program synthesis [AC18] to analyze imperative code in search of equivalent and more performant rewrites (e.g., MapReduce programs). These systems can be used to automatically rewrites UDFs into native equivalents. However, they are generally meant to run offline. For example, despite the fact that Casper [AC18] accelerates the synthesis process by searching over program summaries, it still takes an average of several minutes to run. In contrast, we target online UDF translation.

3.4 Motivating Examples

We present several example queries that 1) expose the limitations of exact-match caching, 2) motivate the need for aggressively identifying result caching opportunities, and 3) explain how supporting UDFs is critical for result caching. The presented queries are based on two hypothetical data tables, `people` and `siblings`, that share the two-column schema, `name` and `age`.

3.4.1 Need 1: Aggressive Identification of Result Caching Opportunities

Spark SQL's exact-match caching (§3.2) that only uses cached results if the corresponding query plans exactly match, foregoes critical caching opportunities. For example, consider the following queries:

```
// Query 1
people.join(siblings, "age").filter(people.age > 18)
// Query 2
people.join(siblings, "age").filter(siblings.age > 21)
```

Both queries perform a `join` on the `people` and `siblings` tables, finding siblings of the same age. Thus, any predicate applied to `people.age` is in effect applied to `siblings.age`; these

two columns can be treated as interchangeable for the rest of the query. Consequently, despite the fact that the two queries employ seemingly different filters (shown in bold)—Query 1 filters the `age` column in `people`, while Query 2 filters the `age` column in `siblings`—Query 2 should be able to reuse results from Query 1. Specifically, since any age greater than 21 is also greater than 18, Query 2's result set will always be a subset of Query 1's result set; that is, Query 1 *totally subsumes* Query 2. Reusing results from Query 1 to compute Query 2 enables an in-memory scan of a (potentially) much smaller table, rather than recomputing the expensive join. However, Spark SQL's cache manager would deem these queries as unrelated because it is unaware of the equivalence relationship that the join creates between the seemingly different filter predicates.

Another result caching opportunity that would go undetected with Spark SQL's exact-match caching relates to the input of a join operation. For instance, consider the following queries:

```
// Query 3
people.filter(age > 18)
// Query 4
people.join(siblings, "age")      .filter(people.age > 18)
```

Query 3 filters all rows that have an age greater than 18 in the `people` dataset. In contrast, Query 4 first joins the `people` dataset with the `siblings` dataset to find rows with matching age values, and then applies the same exact filter. Thus, though Query 3's result set does not entirely contain Query 4's result set, Query 3 does produce a useful input for the join in Query 4. In other words, Query 3 *partially subsumes* Query 4.

To help understand why Spark SQL would fail to detect this relationship, consider the analyzed and optimized query plans for these queries shown in Figure 3.2. Comparing only the analyzed plans of each query (which Spark SQL's cache manager does) hides the fact that Query 3 partially subsumes Query 4. However, during optimization, Query 4's `filter` predicate is pushed down to below the join. This transformation highlights the fact that Query 3 is identical to the left join child in Query 4, enabling caching.

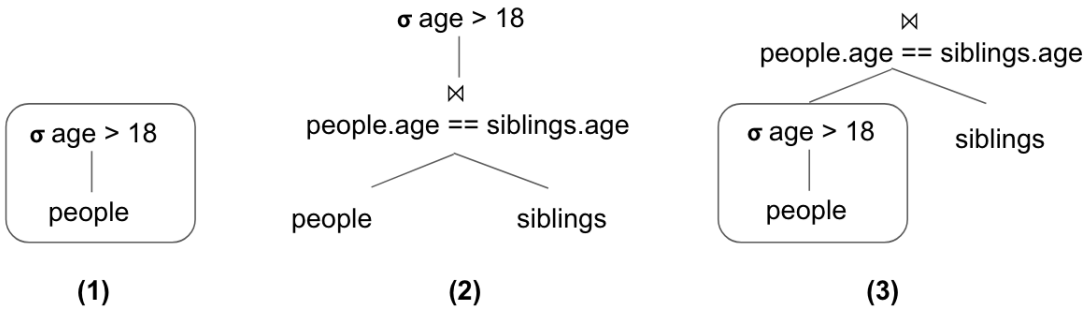


Figure 3.2: (1) The (identical) analyzed and optimized plans for Query 3, (2) analyzed plan for Query 4, and (3) optimized plan for Query 4. Boxes show a cache opportunity: Query 3 can safely be used as Query 4's left join child.

3.4.2 Need 2: Result Caching Support for UDFs

Detecting the caching opportunities between the above queries relies on the ability to analyze query predicates. However, UDFs hide query components from the query engine, obscuring even exact cache matches (§3.2). Consider the following two queries:

```
// Query 5
people.select(lower("name"))
// Query 6
people.map(p => p.get("name").toLowerCase)
```

Both queries return the list of `names`, converted to lowercase format, from the `people` dataset. Indeed, in the absence of null values, Query 5 and Query 6 will always return the same result set, meaning that they are *valid rewrites* of one another. However, from the perspective of caching, Query 5's structure is preferable because it uses a native call to Spark SQL's DataFrame API and can thus be analyzed in detail by the optimizer; in contrast, Query 6 contains a UDF closure that makes an external call to a Scala library, and is thus treated as a blackbox during planning. Simply put, Query 6 uses a UDF while Query 5 does not, and this impacts caching.

3.5 Aggressive Result Caching

This section presents a judicious adaptation of predicate analysis on analyzed query plans that enables aggressive result caching without unnecessary query optimization.

3.5.1 Challenges and Approach

A natural and proven approach for detecting result caching opportunities (both exact match and subsumption relationships) is predicate analysis [GL01,IR95,ZND01,CR00]. With predicate analysis, logical operators are grouped by how they manipulate a dataset (e.g., column-removing versus aggregating). The most restrictive predicates within each group are then identified and compared to infer caching opportunities. Predicate analysis can be used in queries containing selection, projection, joins, grouping, and aggregation.

Spark SQL's use of immutable datasets does provide some advantages to performing predicate analysis compared to a traditional database system. In particular, unlike with databases [GMa, MRS01, ZLE07], we need not worry about stale (and inaccurate) cached data. However, the associated query optimization overheads (§3.1) makes it difficult to efficiently integrate predicate analysis. To illustrate this challenge, we first discuss several potential integration approaches (and the associated consequences), before presenting our solution; we empirically compare these approaches in §3.8.

Swap: Make caching decisions later in the pipeline. The most natural approach is to reorder the pipeline such that cache substitution comes after query optimization (i.e., swapping stages 2 and 3), thereby exposing optimized query plans to predicate analysis. Unfortunately, this would forego query optimization benefits as a query must pass through the entire optimizer before any caching decisions can be made and applied (via predicate analysis and query rewriting). This is despite the fact that optimizations must only be run on uncached query components. We show in §3.8 that these passes through the query optimizer result in significant resource and delay overheads [Mar06, Neu11].

Double: Insert an additional cache retrieval step. In this scenario, the pipeline would have a cache retrieval step both before and after query optimization. Early-stage caching can identify exact matches on analyzed plans, while late-stage caching can use predicate analysis on optimized query plans to more aggressively identify caching opportunities. Thus, this approach partially addresses the limitations of Swap: query components handled by the cache (identified by exact match) need not pass through the query optimizer. However, this approach has several limitations. First, the potential in-exact query matches that predicate analysis identifies (which have proven to be significant in databases and early data analytics frameworks [JMH16, CY12, EA12, NPM10, CCH16, GRT10, JKR18, KFM17]) can still only be determined after costly query optimization, leading to wasted work. Second, it doubles the size of the cache index by storing two or more versions of query plans per cached job. This scales particularly poorly with cached jobs that have many operators—jobs that are the best candidates for result caching.

Our solution: Given these tradeoffs, our solution is to partially optimize analyzed query plans, only enforcing rules that result in canonicalized predicates that affect caching decisions. In this way, predicate analysis can run early in the pipeline and still operate on standardized and information-rich query plans necessary to make aggressive but correct caching decisions. Our key observation is that only a few query optimizations influence predicate analysis (and thus caching decisions), so the cost of this early and partial query optimization is small.

3.5.2 Detecting Subsumption on Analyzed Plans

We begin with the simpler "total subsumption" case where a cached query entirely contains the result set of another query. We then discuss extensions to handle the scenario where the cache contains only part of the new query's results (i.e., "partial subsumption").

Total Subsumption: We first seek to identify the set of optimizations that query optimizers perform which generate canonicalized predicates that affect predicate analysis (and caching decisions). To answer this question, we classify optimizer rules based on how they affect the query

tree. Optimizer rules may *reorder*, *replace*, or *rewrite* operators. Reorder rules do not affect predicate analysis in total subsumption, as all operators are sorted during analysis anyway. Replace rules may help generate efficient plans by swapping operators, but sorting predicates groups equivalent operators and extracts their predicates with the same effect. Instead, rewrite rules yield a standardized structure that is beneficial to predicate analysis, so we extract and apply these.

Analyzing the Spark SQL query optimizer rules revealed that out of the 19 total rules covering 100 structural patterns, only these 4 rules (covering 25 structural patterns) are rewriting rules that standardize operator syntax for predicate analysis:

1. **Typecast checking:** lifts raw values out of cast expressions after confirming that the cast is type safe.
2. **Boolean simplification:** standardizes Boolean expressions, for instance by rewriting `not` operators into equivalent positive expressions.
3. **Constant folding and propagation:** evaluates constants and uses them when possible.
4. **Operand ordering:** standardizes operand orders for quick comparisons.

Thus, we prune the optimization suite to only include these 4 rules and run them iteratively to a fixed point over the relevant predicates in analyzed query plans.

After canonicalizing predicates, we can directly apply existing algorithms to identify total subsumption relationships on our partially optimized query plans [GL01]. Beginning with the cross product of all source tables, the following must be met:

- **Row Removal:** The cached job must remove the same rows (or a subset of them) as the new job. If predicates impose a range on the column, the range must be larger or equal to that of the new job.
- **Column Removal:** The cached job must output all columns needed by the new job; specifically, it must contain the output columns of the new job and any columns needed to calculate

any new predicates.

- **Grouping:** All groupings in the cached job must be supersets of groups in the new job, and the cached job must be less aggregated.
- **Other operators:** Any other operators in the cached job are deterministic, and either invertible or applied to the same inputs in the new job.

To further enhance the detectable caching opportunities with predicate analysis, we identify column equivalence classes [GL01]. Equivalence classes are used to specify that seemingly different columns are equivalent for a given query based on the query's filtering predicates (e.g., Query 1 and Query 2 in §4.2). To build equivalence classes, each column begins in its own class. For every predicate of the form ColA $==$ ColB, the corresponding equivalence classes containing ColA and ColB are merged, since these columns will always have the same value. The above conditions are then checked using equivalence classes in place of columns.

If total subsumption scenarios are detected, Acorn rewrite the new query's analyzed plan to use the relevant cached results. Recall that with total subsumption, the cached results may include extra rows beyond what the new query requires. Thus, we add the required filter predicates from the original plan to remove extraneous rows. Since we only add filters, the full optimizer pass addresses any introduced inefficiencies.

Partial Subsumption: A query A partially subsumes another query B if it totally subsumes a join node in query B (e.g., Query 3 and Query 4 in §4.2). With optimized query plans, this relationship can be found by running the total subsumption algorithm described above on the children of each join node in query B. This would work because optimized plans employ predicate pushdown, whereby predicates that remove columns or rows are pushed below a join if possible (using reorder rules) [HS93].

Predicate pushdown is crucial for detecting many partial subsumption caching opportunities. This is because the smaller a join's child is, the more likely we can find a cached job that subsumes its results. To increase our chance of success, we want to scan the entire query and push

any operators that remove rows below a join if possible. However, even assuming a standard predicate pushdown pass has been made, introducing partial cache subsumption breaks a critical safety guarantee in the Spark architecture. The optimizer assumes all column references have been disambiguated with respect to the base tables of the query; when Acorn replaces a base table with a partial subsumption match, it assumes responsibility for ensuring that this disambiguation remains faithful to the original tables. For example, consider the following query pair:

```
val q1 = people.select("name")
val q2 = q1.filter("age" > 21)
```

Notice that q2 references `age`, which is not in `q1's` output schema; Spark allows users to reference attributes that are not in the projection list because they will be resolved during reference disambiguation. This can cause subtle errors when caching plans. For instance, consider the following cached plans:

P1: `people.select("name").filter("age > 18")`

The semantics of P1 are such that the filter clause can be evaluated. However the cached output does not contain `age`. Thus, P1 is *not* suitable to replace q1 in q2's plan, because `age` was not materialized. By default, Spark's reference disambiguation will incorrectly infer that`age` is available in `people.select(``name'')` of the cached plan P1.

To address this limitation, we present an algorithm for detecting partial subsumption on the children of joins in analyzed query plans. Our approach uses a predicate sorting technique that mimics the effect of predicate pushdown while simultaneously guaranteeing safe accesses to all column references—neither predicate pushdown nor Spark's projection analysis can achieve both in a single pass.

Since we want to maximize constraints on each join child, we push down all filter predicates that reference any column in the same equivalence class as a column originating from that join child. At a high level, we tag all equivalence classes necessary to compute each predicate in the entire query, then delete a predicate if the join child cannot supply a column in each equivalence class.

To carry out this approach, we first need to know, for each predicate P that appears anywhere in the entire query, the set of all columns required to calculate that predicate (which we call the $refset$). We compute $refset$ by initializing it to the empty set and adding all equivalence classes (EC) of each column directly referenced in that predicate,

$$\text{refset(P)} \leftarrow \text{EC}_1, \text{EC}_2, \ldots \text{EC}_N,$$

where $1...N$ are ids for the columns referenced in P.

With this information, we can discard predicates that cannot be computed by the base tables of this join child. To do this, we take the set of all columns that appear in a base table of the current join child as the child column set (CCS). We then keep only predicates if its refset consists only of equivalence classes with some column in the child's base tables:

$$\{\text{P} \mid \forall\ \text{EC} \in \text{refset(P)}: \text{EC} \cap \text{CCS} \neq \emptyset\}.$$

The above steps tell us what row-removing predicates we should keep. We also must know what columns to keep, which we can find by using the refset: any column appearing in both the *CCS* and any *refset* is required as output from the join child, and thus must be kept.

Once we have pushed down all the appropriate filter predicates and calculated the new output set for a join child, we can perform total subsumption analysis on that join child to find caching opportunities. Rewriting on a cache hit uses the same steps as with total subsumption.

Correctness: We meet both conditions proven to preserve correctness when moving predicates [HS93]. First, our approach does not change the order of join operations. Second, we ensure that every root-to-leaf path in the query tree only refers to attributes produced by the join child. This follows from the fact that, by definition, the *refset* contains all column references in a predicate. Cross referencing the *refset* with the *CCS* ensures that a predicate is only moved down if it can be independently computed by the join child.

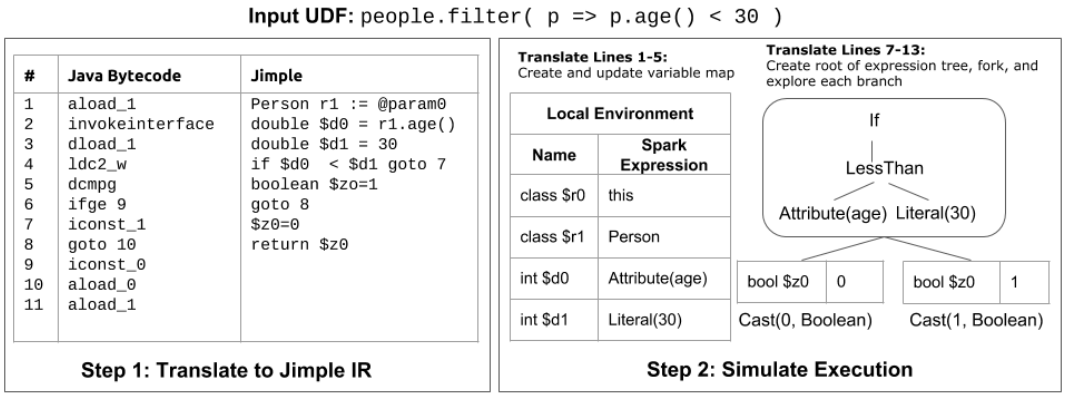


Figure 3.3: The translation steps used to convert a simple UDF into a native Spark Expression. Code segments and expression trees have been trimmed due to space constraints.

3.6 Transparent UDF Compilation

This section discusses our approach to transparently open up UDFs (written in a general purpose language) to query planners to enable the aggressive result caching techniques presented in §3.5.

3.6.1 Goals and Solution Overview

We have several goals and requirements for UDF translation that existing techniques do not meet (§3.3):

- **Transparency:** Users should not have to rewrite queries, annotate jobs for the sake of the optimizer, or restrict themselves to only using registered UDFs.
- **Speed:** Translation overheads must be lower than optimization benefits.
- **Safety:** Translated expressions must behave exactly the same as the original expressions *on all inputs*.
- **Tractability:** Translating general purpose languages like Java and Scala requires parsing Java bytecode, which is a stack-based language with over 200 opcodes. Thus, the search space is large and must be explored efficiently.

- **Extensibility:** A UDF translator should be easily maintained and extended to support new or modified language features.

While prior work meets some of these goals, no current solution satisfies all of them (§3.3). UDF rewriting [PMZ, PTN18] and annotation techniques [LSH14] violate our transparency requirement, while program synthesis approaches [AC18] take minutes to run. Further, UDF compilers like Froid [RPE17] only operate on constrained languages (T-SQL), and cannot support many general purpose language features (e.g., generics, reflection).

Our solution: We provide a best-effort UDF translator for general purpose languages that meets all of the above requirements. The translation process passes each UDF through three steps. First, we use an off-the-shelf bytecode parser to generate a typed, compact intermediate representation (IR) for the UDF. We then use symbolic (simulated) execution and pattern matching to quickly generate an equivalent query plan expressed solely with native query operators and API calls. Finally, we optionally rewrite the UDF if compilation is successful.

Using a compact and typed IR significantly trims the search space to consider, ensuring tractability and fast translation. Simulated execution ensures that each unique path through the function is explored exactly once. Importantly, our simulated execution propagates types from the IR (rather than inferring them). Collectively, these techniques ensure a type-safe, accurate, and efficient translation. Further, because our translation leverages Scala's pattern matching capabilities [KM18], new operators can be supported with a single line of code (for a new pattern), bringing extensibility. Our integration of this translation into query processing pipelines (§3.7) makes this entirely transparent to users.

3.6.2 UDF Translation

Our translator accepts a Scala or Java function as input, and outputs an equivalent expression solely with Spark native operators. To aid our description of the translation process, we will reference the example command in Figure 3.3: `p => p.getage < 30` is a Scala lambda UDF that is passed

as an argument to the `filter` operator, and applied to a dataset named `PersonDS`.

3.6.2.1 Step 1: Lower to an Intermediate Representation

The translator initially receives the UDF in Java bytecode form. Raw Java bytecode is an stack-based language with over two hundred opcodes and bytecode specific types. Thus, simulating execution of Java bytecode directly would require the burdensome and error-prone tasks of maintaining a stack and inferring types. To obviate the need for stack simulation or dynamic typing, we instead opt to perform translation directly on an IR called Jimple [VH98]. Jimple reintroduces high-level types and provides a concise, three-address format with only a few dozen opcodes. Reintroducing types ensures that the translator must only track types during compilation (rather than inferring them) for type safety, while the three-address format obviates the need for simulating a local variable stack.

Though translation with the Jimple IR greatly simplifies the process compared to working with Java bytecode, we note that using an IR is not strictly necessary for the ensuing steps. In order to compile down to Jimple, we use an off-the-shelf bytecode parser called Soot [VCG10], which adds minimal overheads to the overall translation process (§3.8). Figure 3.3 shows a comparison of our example lambda (written in Scala) with its Java bytecode and Jimple equivalents.

3.6.2.2 Step 2: Simulate Execution

The next step compiles the UDF's Jimple representation into a Spark native expression. Our compiler accepts as input the Jimple function, the function arguments, and a schema for the corresponding datatype. Spark SQL allows schemas to be defined as classes where each field is a column, so the schema input may be a class definition or a struct pairing column names with types. For example, the function from Figure 3.3 would be supplied with the argument for `p` and the `Person` class schema.

Translation progresses by simulating execution of the Jimple function body. We use three tech-

niques to explore the search space safely, efficiently, and comprehensively. First, we use a map to keep track of a typed local environment. Types are propagated to the map from Jimple throughout translation, guaranteeing a type-safe translation. Second, we use Scala's pattern matching capabilities to facilitate translation from Jimple into a Spark expression. Finally, we use reflection to examine function signatures to decide whether to substitute recognized library functions with Spark expressions or recursively translate the function call.

Local environment: To mimic a local environment, we use a map (rather than a stack) to pair UDF variable names and types with Spark expressions. Assignment statements do not have a corresponding Spark expression, and are instead used to populate or update the map. The right-hand side of each assignment statement is translated and the resulting expression is placed in the map. For instance, if the right-hand side is a constant or a column (as in line 3 of Figure 3.3), we map the value into a Spark literal of the corresponding type. Similarly, if the right-hand side is an expression, it is translated into the corresponding Spark expression and the operands are then recursively translated. When any variable is read during subsequent translation steps, a typed Spark expression value is supplied from the environment map.

Translation: Jimple code is organized as a series of statements, each of which is composed of expressions, which are in turn composed of values. Spark expressions, our target language, are structured as trees. To match Jimple expressions by type and create the corresponding Spark tree node, we use Scala's pattern matching capability [KM18]. Any Jimple subexpressions or values are extracted via pattern matching, recursively translated, and added as children. This flow enables subexpressions (e.g., child expressions) to be translated independently of higher level expressions (e.g.,parent expressions). Further, adding a new operator only requires matching the corresponding Jimple expression and creating the corresponding Spark expression tree node, which can be expressed in a single line of code.

Path exploration: Simulating UDF execution requires systematic exploration of each path through the function. In a function with only a single path, every Jimple statement creates or updates a variable until a `return` statement is reached. Jimple always returns a variable, so the translator

retrieves the associated expression from the local environment map and returns it.

The more common scenario is for a function to contain multiple execution paths, which arise from conditional statements (e.g., line 4 of Figure 3.3). In this case, the conditional predicate is translated, and the translation process forks to find an expression for each branch separately. Special care is given to short-circuiting predicates as Spark SQL does not enforce the short-circuiting semantics of Java and Scala logical operators. To address this, `Acorn` uses conditionals to mimic short-circuiting behavior: short-circuiting `AND`s are rewritten to be nested `if` statements (e.g., ``` ``if x != null && x.a'' ``` would be rewritten to ``` ``if x != null { if x.a }'' ```), while `OR`s are broken into `if-elif-else` blocks.

The first conditional statement encountered during translation is considered the branching point, and thus represents the root of all paths through the function. A branch is explored by providing the translator with the relevant variable environment, original function arguments, and a pointer indicating the first instruction of the branch. The translator executes from that instruction until it reaches a return. To ensure accuracy, conditional variable assignments are preserved; branches maintain their own copies of the environment to prevent conflicts.

Function calls: Some expressions may invoke a function, like `r1.age()` in line 2 of our example. In this case, we either invoke the translator and explore the provided `age` function or use reflection to examine the function signature. The latter approach is preferred as it lets us bypass the translator and directly supply the matching Spark expression. In our example, we would match the `age` function with the `Person` class schema to determine that the function results in a column reference. Since Spark operators usually handle `null` inputs quietly while library functions throw exceptions, we are careful to reintroduce `null` exceptions where necessary. In cases where the function signature is unknown but lies on the user class path, we attempt to translate the function to find a matching Spark expression. This allows us to handle important library functions while limiting expensive library function execution.

3.6.2.3 Step 3: Rewrite the UDF

On successful translation, the UDF query tree should be rewritten to use the output Spark expression. Since query plans use a tree structure, rewriting only involves replacing the UDF node with the root of the new Spark expression. This can happen in one of two ways. First, if a UDF node appears directly as an operator, we simply replace the UDF node in the expression tree with the root node of the generated Spark expression. Second, a UDF may be the argument to another operator. In cases where the operator can take a function or a Spark expression as its argument (e.g., `filter`), we translate the function and swap the parent operator for a version that accepts a Spark expression. For operators that only take functions as arguments (e.g., `map`), we add another version of the operator to Spark (that accepts Spark expressions as arguments) and use the same process. If our translator cannot generate a native Spark equivalent (§3.6.3), we revert to the original UDF path to ensure correctness; failed translation overheads are negligible (§3.8).

3.6.3 Correctness and Limitations

Our translator supports almost all features of the Java and Scala languages, and is primarily limited by the target language (i.e., Spark's native API). We handle variable manipulation, control flow statements, and logical, bitwise, and arithmetic operations. Additionally, unlike Froid [RPE17], we preserve types and can support UDFs that use generics, reflection, and virtual function invocations. The key limitation is that we are restricted to non-recursive function calls and statically bounded loops. The impact of this limitation is mitigated by Spark's requirement that query trees be acyclic and its corresponding lack of support for loops. Note that we can translate loops that are unrolled by the Java compiler or contained in a library function invocation. Examples of the latter class include most String and Array manipulations, for which we translate based on the function signature. Additional library function invocations we support are type casting, column/row access, and sorting. Nondeterministic functions (e.g., `Random` class functions) cannot be translated as they complicate faithful simulated execution.

3.7 Implementation

`Acorn` integrates our aggressive result caching optimizations (§3.5 and §3.6) into the query processing pipeline (Figure 3.1) of the most recent version of Spark SQL (v2.4). As Spark itself is written in Scala, so is `Acorn`. We note that Acorn does not increase the amount of cached content (or memory usage), and instead only tries to make better use of what Spark already caches.

3.7.1 Judicious Predicate Analysis

`Acorn` implements predicate analysis in Stage 3 (cache substitution) of the Spark SQL pipeline. To do this, `Acorn` edits the base class for analyzed plans, augmenting them with functions for containment detection and rewriting. `Acorn` also modifies Spark SQL's cache manager; instead of using exact matching, the manager calls `Acorn`'s custom containment function which implements subsumption detection (§3.5) in 250 lines of code. If a cache opportunity is detected, this function rewrites and returns the query accordingly.

3.7.2 Transparent UDF Translation

UDF translation is structured as a standalone unit (400 lines of code) to simplify integration into the pipeline. UDFs that appear as the argument to an operator are translated when the operator itself is parsed. The operator is then rewritten to use the generated Spark expression instead of the UDF. This happens just before stage 1, when the query is parsed and transformed into an input for stage 1. For UDFs that act as an independent operator, `Acorn` adds a rule to the analyzer to translate and rewrite the UDF during stage 1. Translation happens before cache replacement in Stage 3 so translated UDFs undergo the caching optimizations described in §3.5.

3.7.3 Generalizing Beyond Spark SQL

Although Acorn is implemented in Spark SQL, other data processing systems can naturally benefit from its optimizations. Systems that struggle to reuse cached plans (e.g., SQLServer [Mar06]), especially those that use immutable datasets (e.g., CouchDB [cou], Datomic [Kie13], BigTable [CDG08]), can perform partial query optimization to use predicate analysis without unnecessary query optimization. Acorn's partial optimization rules are Spark-specific, but the approach to identify necessary rules for partial optimization generalize (i.e., the grouping of rule types in §3.5.2). Similarly, Acorn's UDF translation is JVM-specific as it operates on Java bytecode, but the lowering and UDF analysis can be implemented for Python or applied to other multi-language pipelines (e.g., Pandas [McK11], DyradLINQ [FBE09]) whose UDFs can be compiled to an IR (e.g., asm, .NET, LLVM).

3.8 Evaluation

In this section, we experimentally evaluate Acorn and find that 1) Acorn can significantly accelerate workloads compared to Spark SQL, with benefits of 2× and 5× for benchmark workloads with and without UDFs, respectively; 2) Acorn outperforms the alternatives for predicate analysis described in §3.5 by avoiding unnecessary query optimization; 3) Acorn's benefits range from 1.4×–3.2× for real graph algorithm workloads; 4) Acorn can translate 90% of UDFs collected from multiple real Spark workloads, many of which Froid [RPE17] cannot; and 5) overheads for Acorn's predicate analysis and UDF translation techniques are negligible.

3.8.1 Methodology

We evaluate Acorn on three main workloads:

TPC-DS v2.1: From this big data benchmark [tpc], we use all 4 queries marked as "iterative," which each come with 2-5 variants. We also select 10 random "reporting" queries which

are parameterized by a random number generator; we create multiple variants by resampling the parameter values. In total, we use 14 base queries from TPC-DS with 28 variants, for a total of 42 queries. To create UDF versions of the queries, we convert the SQL string to an equivalent query that uses the Spark SQL DataFrame API (§3.2). We mark the first executed variant of each query for caching, allowing queries later in the sequence to reuse previously cached queries; the cache is cleared between runs.

TPC-H: For this benchmark [tpc], we use all 22 queries. These queries directly include 7 UDFs, and we augment this list with the 12 additional UDFs used in the evaluation of Froid [RPE17]. In total, this workload includes 19 UDFs and 34 UDF invocations; we manually rewrote each UDF into an equivalent Scala version. We use dataset sizes of 1, 10, and 100 GB for this and TPC-DS.

Real-world workloads: We use two graph algorithms from the GraphFrames Spark graph processing library [gra]: connected components and belief propagation. Belief propagation contains one UDF to color the graph, while connected components has none. For datasets, we use publicly available snapshots of graph data from the SnapNet project [LK14]: a snapshot of the Twitter network from 2010 with 41.6 million vertices and 1.5 billion edges, and a snapshot of the Berkeley-Stanford web graph with 700K vertices and 7.6 million edges. In addition, to evaluate `Acorn`'s UDF translation, we extract real UDFs from seven open-source repositories. [sry, men, Nic, Ryu, biy, Abh, nod]

We compare five systems. Our *Baseline* is unmodified Spark (v2.4), and we consider two versions of `Acorn`: *`Acorn` _cache_only* performs predicate analysis-based cache detection, while *`Acorn`* also uses UDF translation. In addition, we implemented and evaluate the alternative predicate analysis approaches described in §3.5: *Swap* and *Double*. Each workload is run five times with each system, with the cache cleared between runs, and we report on the overall distributions. Tests were conducted on a 16 machine cluster, where each machine ran Ubuntu 12.04 and had an i7-4770 processor, 32 GB of RAM, and 1 TB disk.

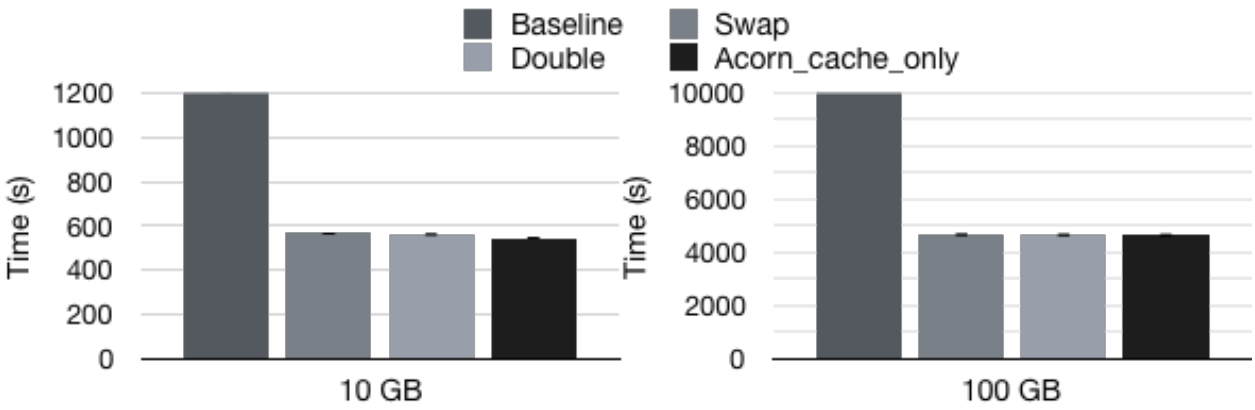


Figure 3.4: Evaluating `Acorn` on the SQL-only (no UDFs) TPC-DS workload. Bars represent median workload completion times (i.e., summing across all queries in the workload), with error bars spanning min to max.

3.8.2 Aggressive Caching with `Acorn`

Here we evaluate `Acorn`'s caching, without considering UDF translation (i.e., `Acorn_cache_only`).

Speedups: Figure 3.4 shows that subplan caching improves runtimes by 2.2× for the entire 1GB and 10GB TPC-DS workloads (161s, 662s saved respectively), and up to 2.7× for the 100GB dataset (5942s saved). Since this workload doesn't contain UDFs, Acorn, Swap, and Double identify caching opportunities for the same 26 queries (69%), compared to 14 (33%) for the baseline.

The larger improvements on the 100GB data are because the queries become disk-bound. On smaller datasets, the data is read into memory once and used to serve all queries against that data. For larger queries, the entire dataset must be paged into memory for each query since it cannot entirely fit. Therefore, larger datasets see more benefit from caching since in-memory relations can be used rather than reading from disk.

Sample TPC-DS queries: Figure 3.6 lists several example queries from the TPC-DS benchmark which illustrate various subsumption relationships which `Acorn` can identify and exploit. The first segment shows the SQL syntax for Q39a and Q39b. As shown, Q39a omits the last predicate, commented in red, which is present in Q39b—this represents a common total subsumption

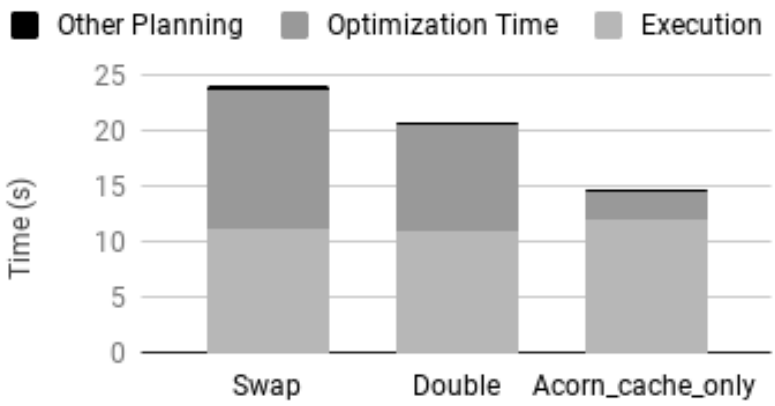


Figure 3.5: Breaking down `Acorn`'s execution, compared to Swap and Double, on the 29 queries in the TPC-DS workload which automatically used some cached data. Results are for the 10 GB dataset, and times are the median of five runs.

pattern in which an additional predicate is applied to the result of a previous query. Acorn is able to recognize this total subsumption relationship and reuse the results of Q39a, unlike baseline Spark.

Q23a includes a `union all` operator over two subqueries; note that the subqueries are condensed for brevity as `Subplan A` and `Subplan B`. Q23b calculates an aggregate over one of the output columns of Q23a, again demonstrating a total subsumption relationship. Q23c adds new predicates to both `Subplan A` and `Subplan B`. Unlike baseline Spark which re-executes the entire query, Acorn can reuse the subqueries from Q23a or Q23b.

Finally, the last example demonstrates how TPC-DS query parameters are re-rolled to generate alternative versions of reporting queries. The Q37 template generates new queries by randomly picking the parameters labeled (i), (ii), and (iii). There are three possible cases for reuse across rolls. For total subsumption, (i) must exactly match, and the second rolls of (ii) and (iii) must either match the first rolls exactly or produce subsets of the first roll values. For partial subsumption, either (i) must be an exact match and (ii) must be a subset in which case the `item` table can be reused, OR (iii) must be a subset in which case the `inventory` table can be reused. Acorn is able to detect all three subsumption scenarios.

```
q39a, 39b
select warehouse_name,warehouse_sk,item_sk,d_moy,
       inv_quantity_on_hand
from inventory, item, warehouse, date_dim
where inv_item_sk = item_sk
         and inv_warehouse_sk = warehouse_sk
         and inv_date_sk = date_sk
         and d_year = 2001
         and d_moy = 1
 and inv_quantity_on_hand > 2                              ---diff---
```

```
q23a, 23b, 23c
select c_last_name, c_first_name, sales from
(SUBPLAN_A )
union all
(SUBPLAN_B))

select sum(sales) from                                     ---diff---
(SUBPLAN_A )
union all
(SUBPLAN_B))

select sum(sales) from
(SUBPLAN_A where cs_bill_customer_sk =
    cs_ship_customer_sk)                                    ---diff---
union all
(SUBPLAN_B where ws_bill_customer_sk =
    ws_ship_customer_sk))                                   ---diff---
```

```
q37
select i_item_id ,  i_item_desc ,  i_current_price,
    inv_quantity_on_hand, i_manufact_id
from     item, inventory, date_dim, catalog_sales
where i_current_price
    between X AND  X + 30                    --- (i) must be exact match
and   inv_item_sk = i_item_sk
and   d_date_sk=inv_date_sk
and   d_date between Cast('1999-03-06' AS DATE) and
           ( Cast('1999-03-06' AS DATE) + INTERVAL '60' day)
and   i_manufact_id in (843,815,...)         ---(ii) list must be subset
and   inv_quantity_on_hand
    between Y and Z                     --(iii) range [Y,Z] must be subset
and   cs_item_sk = i_item_sk
```

Figure 3.6: A selection of TPC-DS queries with differences highlighted in red on commented lines. These examples illustrate several caching opportunities missed by baseline Spark but detected by Acorn.

Benefits of partial query optimization: `Acorn`, Swap, and Double identify the same caching opportunities in TPC-DS. However, Figure 3.5 shows that Swap and Double can nearly double the final runtimes of queries served from the cache due to excessive optimization overheads. The reason is that both Swap and Double force some cached queries through the full optimizer: Swap

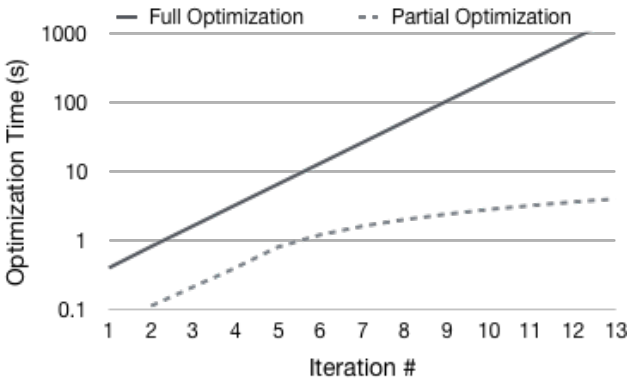


Figure 3.7: Spark SQL's full optimization (Baseline) versus `Acorn`'s partial optimization on the connected components algorithm. Note that the y-axis is logarithmic.

fully optimizes all queries to find exact or inexact cache matches, while Double fully optimizes queries without exact matches (including those with inexact matches). The median cost of this unnecessary optimization for Swap and Double is 0.43ms per query (12.5s total). Since the median query runtime is only 0.37ms, overheads of this optimization vary from 100–1000% of the overall query runtimes.

High optimization costs are particularly relevant for iterative workloads (e.g., graph processing, ML training) that append to larger and larger plans each iteration. Using the plan after each iteration of GraphFrames' connected components algorithm (§3.8.1), the full optimization cost grows exponentially whereas `Acorn`'s partial optimization grows linearly (Figure 3.7). The same trend holds for belief propagation, which we omit for space.

Cache search overheads: We now study the overhead of using predicate analysis to search the cache by rerunning the TPC-DS iterative workload on a 10 GB dataset. In this experiment, we filled Spark SQL's cache index with all 104 TPC-DS queries and measure the time spent in the search algorithm. The total time spent in search was 10.42s (.1s per query), in part because the matching algorithm can terminate early at multiple places, such as comparing base tables. Note that although the search time is runtime agnostic, this implies an overhead of $<$1% of execution time.

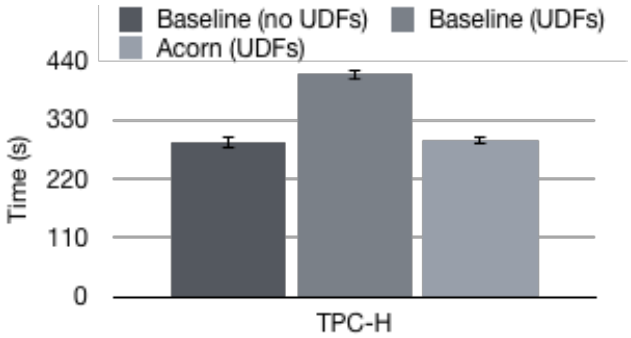


Figure 3.8: Baseline Spark versus Acorn on the TPC-H workload (scaled to 10 GB), with and without UDFs.

3.8.3 Acorn's UDF Translation

We now evaluate Acorn's transparent UDF translation.

Cost of UDFs: Prior work in Froid [RPE17] showed that UDFs incur a 100-10,000× slowdown compared to native relational operators. Thus, we first measure the overhead of using UDFs. Figure 3.8 compares the overhead of using UDFs by comparing baseline Spark on the 10GB TPC-H benchmark without UDFs, with relational operators replaced with Scala UDFs as in Froid, and Acorn on TPC-H with the same UDFs. TPC-H is non-iterative and no queries are cached, thus the differences are due to opening the UDFs to the optimizer. Using UDFs slow baseline Spark by 1.4×, whereas Acorn translates 100% of the UDFs, eliminating serialization overheads and bringing Acorn's performance in-line with the pure relational version. Per-operator UDF overhead was 80-100×—we speculate that the discrepancy with Froid may be due to Spark's efficient Kryo serializer, our use of a 16-machine cluster rather than a single machine, and that SQLServer's native operators are faster than Spark's.

Translating real UDFs: We extracted and ran the UDF translator on 30 Scala and Java UDFs from seven open-source repositories [sry,men,Nic,Ryu,biy,Abh,nod] that contained Spark applications. In total, Acorn translated 27 of them (90%). Two of these could not be translated because the UDF performed some form of I/O which cannot be expressed as a native operator, an inherent limitation of translation. The third required translating a call to an unrecognized, external library function;

Figure 3.9: Original and `Acorn`-translated UDFs from real-world Spark workloads.

although `Acorn` can perform the translation, we are cautious and disallow it since it may lead to loading large library binaries and cause JVM memory contention during translation. **Example UDF translations:** Figure 3.9 shows three example UDF translations with `Acorn`. The first is a UDF (written in Scala) taken from the TPC-H workload. The UDF includes common programming language mechanics such as variable declaration and short-circuiting boolean logic. As discussed in §3.6.2 and shown in the example, Acorn circumvents lack of short-circuiting recognition by breaking conjunctive logic into nested `if` statements. We note that state-of-the-art translators like Froid [RPE17] *can* translate UDFs with such features.

The second and third UDFs contain examples of language features which Froid *does not* sup-

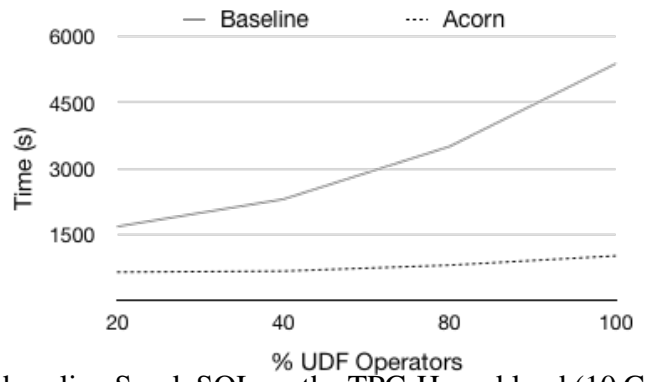


Figure 3.10: Acorn vs baseline Spark SQL on the TPC-H workload (10 GB dataset), with varying fractions of query operators being replaced by UDFs.

port. The second is a closure that, when applied to a typed DataFrame, extracts a class field named birth as a column and performs a simple filter. Note that the translation of this UDF depends on the DataFrame to which it is applied, highlighting the importance of dynamic translation based on the currently loaded class definition: if the DataFrame schema or class file does not have an accessible class field named birth, the UDF will raise an error and Acorn's translation will change accordingly. The third UDF takes a row from a DataFrame, splits it according to a regular expression, and then creates an object instance using the result.

Froid has no clear mechanism for translating object-to-relation constructs such as the class field extraction in the second UDF and the object created in the third UDF. In contrast, Acorn leverages its ability to encode an entire table definition as an expression and use of object reflection to translate both UDFs.

Translation overheads: Translating UDFs with Acorn is a multi-step process. To understand the associated overheads, we test a version of Acorn that performs all of the translation steps other than rewriting. We evaluate this version of Acorn on all 24 UDFs in the TPC-H workload (10 GB), and observe that the total translation time is 540 ms, or 22.5 ms per query, which is well within the margin of variability for workload completion time. Indeed, median workload completion time with this version of Acorn was still 872 ms *faster* than the baseline due to normal variance.

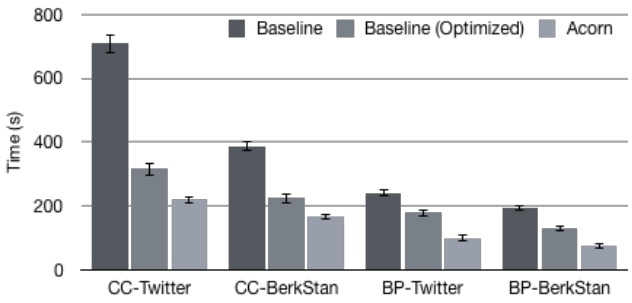


Figure 3.11: `Acorn` vs baseline Spark SQL on two graph processing algorithms: connected components (CC) and belief propagation (BP). `Baseline (Optimized)` manually forces materialization of intermediate data structures. Experiments used snapshots of a Twitter follower graph (Twitter) and the Berkeley-Stanford web (BerkStan) [LK14].

3.8.4 `Acorn`: Putting it all Together

We next investigate the effects of combining `Acorn`'s predicate caching and UDF translation.

Benchmarks: In this experiment, we replace relational operators in the TPC-DS benchmark with UDF versions of those operators. To investigate how performance changes with different workload properties, we vary the percentage of operators replaced with UDFs from 20%–100%. As shown in Figure 3.10, `Acorn` significantly improves performance over the baseline. Speedups with `Acorn` were 2.6× with 20% of operators replaced, and sharply increased to 4.35× and 5.3× for 80% and 100% replacements, respectively. The reason is that, unlike the baseline, `Acorn`'s performance remains mostly flat as more UDFs are introduced, since it can translate those UDFs and find reuse opportunities.

Graph algorithms: We also evaluated `Acorn` on two popular graph processing algorithms: connected components and belief propagation (§3.8.1). As Spark SQL's poor optimization performance is well known, a common "hack" to subvert this inefficiency is for workloads to force materialization by caching datasets, converting them to a (now deprecated) internal data structure, and converting them back. Compared to the optimized baseline, `Acorn` provides median speedups of 1.4× and 1.7× for connected components and belief propagation (Figure 3.11); `Acorn`'s speedups

are 2.3× and 3.2× over the baseline—all without any developer intervention.

3.9 Summary

The benefits that modern data analytics frameworks have achieved by using immutable datasets and increased support for UDFs have come at the cost of suboptimal result caching. This project presented two novel techniques to efficiently enable aggressive result caching in theese frameworks. First, we described a judicious adaptation of predicate analysis on analyzed query plans that avoids unnecessary query optimization. Second, we presented a UDF translator that transparently compiles UDFs, expressed in general purpose languages, into native equivalents. Experiments with our implementation of these techniques, `Acorn`, on several benchmark and real-world datasets revealed speedups of 2×–5× over Spark SQL. Though our implementation and results use Spark SQL, the underlying techniques generalize to other distributed data processing systems.

CHAPTER 4

Jade: A Physical Visualization Design Tool

4.1 Overview

Interactive visualizations are powerful tools for exploring and understanding large data sets. However, building these applications is difficult and resource intensive; not only do application developers need to design a functional front-end visual interface that expresses the appropriate set of tasks, they also need to ensure that the workload expected by the application can be executed by the back-end database at interactive speeds.

This task is particularly difficult for several reasons. First, interactive applications have strict latency requirements; human perceptual and cognition studies clearly demonstrate that both user experience and quality of analytical insights suffer with increased latencies. Moreover, additional studies show that certain types of interactions (i.e. interactions that perform different analytical tasks) are even less latency tolerant than others. In sum, interactive applications demonstrate a workload with strict latencies set at task-level. Second, interactive applications consist of both a client and server component. Clients run the visual side of the application and are capable of performing light processing, while servers can offer higher storage and processing capacities at the cost of incurring a network delay. An application developer must make judicious use of these two components. Finally, as we demonstrate using the commit history of a real-world interactive application, interactive visualizations are often developed iteratively as both desired analytic tasks and data characteristics evolve. This means that the arduous work of developing an architecture that can support interactive speeds must be continuously re-evaluated as the demands of the front-

end application change; if those demands exceed the limits of the current set up, then significant effort must be put into overhauling the supporting architecture.

There is a clear need for a design tool that can automate much of this work, as exists in other analytical execution systems. At a high level, the problem description is *Given an input workload, latency constraints, and a set of resources, what is a sufficient architecture for executing this workload?*. This formulation parallels that of physical database design tools, which select a set of indexes and material views to support a target workload making physical database design (PDD) a natural point of comparison for designing a physical visualization design tool (PVD). Here, we highlight how PVD differs from PDD in several key ways.

- **Strict latency requirements.** As mentioned, landmark cognition and human perception studies [LH14, BHP97, CMS99, CMN83, New90] have concluded that perceived delay enormously impacts the quality of user experience during analytical tasks. Liu et al [LH14] quantified this effect in the context of interactive visualizations, with key findings indicating that response times over 1000ms represent significant degradation for most analytic tasks, with some classes of knowledge discovery tasks demonstrating significant degradation after 100ms. Thus, unlike the problem formulation in PDD, which seeks to minimize overall costs, PVD must optimize to meet constraints on the order of hundred of milliseconds to prevent noticeable performance degradation.

- **Granularity of optimization.** Additionally, the same body of work finds that some types of interactions (e.g. zooming, clicking on a link) are less sensitive (i.e. more tolerant) of delays than others (e.g. hovering for details, adjusting a slider). Given a limited resource budget, more sensitive interactions such as the latter ought to be prioritized by a physical design system. However, PDD optimizes at the granularity of queries; in PVD, where a single query can be shared by multiple interactions, optimization needs to happen at the granularity of interactions. This type of conflicting latency constraint can be met bluntly by forcing an entire query to perform at the most restrictive latency constraint. However, this approach

is often resource intensive to the point of being infeasible; instead, it is necessary to use techniques such as split execution that can refactor a single query into multiple execution plans, each meeting the latency constraint of its constituent interactions.

- **Architecture** Third, existing PDD tools manage data structures that are stored in the back-end database.By contrast, interactive visualizations are often designed as applications that run either in-browser or natively with support from a server. Thus, PVD needs decide optimum placement of a data structures between the client and the server, accounting for both computational cost and the associated cost of data transfer.

- **Diversity of physical structures.** Finally, the configurations suggested by PDD tools consist of a set of columnar indexes and materialized views. However, visualization applications require a more diverse set of data structures in order to meet its strict latency requirements, including spatial indexes, pre-aggregations, and multi-dimensional indexes. The set of data structures considered by a physical design tool impacts both the size of the search space and the analysis techniques used to suggest data structures for a workload.

Based on our observations about the differences between these two domains, we propose a new approach to the key components of a physical design tool targeting the interactive visualization domain.

First, we define a novel workload representation, called `Difftrees`. Chen et al [CW20] first introduced tree-like structures that compress multiple potential abstract syntax trees into a single tree using *choice nodes*. In our work, we define a formal semantics for choice nodes, integrating them into the relational algebraic trees used during physical planning. Our semantics build on that of *parameterizations* already in use in applications such as SQLServer; however, while parameterizations are limited to replacing literals in selection predicates (replacing the expressions e.g. `state = "HI", state = GA"` with the single expression `state = @param`, choice nodes are more powerful because they can represent structural (i.e. non-literals) transformations, even if they don't share a similar grammatical structure (e.g. combining `state = "HI"` and

`state NOT IN["HI", "AL"]` with the single expression `ANY{state NOT IN["HI", "AL"], state = "HI"}`.

This has several benefits beyond simply condensing multiple queries into a single expression. Because choice nodes have a precise semantics within relational algebra, the optimizer can leverage transformations over those choice nodes to uncover patterns in the workload that can be better optimized. This parallels the benefit of parameterized queries, which support optimization in PDD by revealing which columns are most suitable for indexing; similarly, choice nodes support optimization for other data structures such as pre-computed aggregates or sliced cubes in PVD.

Additionally, choice nodes are powerful because they introduce a user-level grammatical feature against which latency expectations can be specified. Since updating a choice node (or group of choice node) corresponds directly to an interaction in application-level user space, developers can specify that all queries updating a choice node need to run within a specific limit. By decoupling latency specifications from entire queries and replacing them with choice nodes, the optimizer can judiciously manipulate difftrees to isolate latency-intolerant choice nodes and optimize their execution plan.

As our second contribution, we formalize physical visualization design as an optimization problem with unique characteristics. We incorporate hard constraints at the granularity of interactions and physical placement as variables within the optimization problem. We empirically show that the search space is prohibitively large, even for small workloads. We then provide two novel techniques for solving it.

First, we leverage the precise semantics of choice nodes to introduce transformation rules that allows the optimizer to use standard rule-based transformations to find potential execution plans. We then avoid the overhead of exhaustive search by integrating heuristic search within a standard-rule-based optimizer. Heuristic search systematically compares the list of available data structures against the list of available transformations transformations, pruning unreachable data structures. Empirically, this allows our tool to rapidly navigate the search space of candidate data structures without sacrificing quality of results.

Second, we make use of an key insight to simplify architecture selection. Since latency expectations are a hard constraint, there is no marginal benefit between two execution strategies that both satisfy the same constraint even if one is significantly faster. This insight allows us to search for a configuration by only exploring tradeoffs in resource consumption on the client and server (i.e. the amount of physical memory required in each location by an execution plan). We use this observation as a basis for an alternate architecture selection strategy, in which we use the set of Pareto optimal execution plans to successfully generate an application-wide architecture in a non-exhaustive search.

4.2 Motivating Example

A major challenge that developers face when creating interactive visual interfaces is handling the alterations that occur over time to either the interface's design and/or the underlying dataset. The difficulty stems from the fact that, to meet performance and functionality expectations, both the interface and system design must be kept in sync with one another throughout the creation and updating processes. We illustrate the challenges of interface and system co-design using the New York Times (NYT) US Covid visualization[1] as a running example. The discussion is based on our analysis of the commits in the interface's public Github repository.[2]

Interface Description: At the onset of the COVID-19 pandemic, news agencies including NYT designed and continually redesigned interactive visualizations of local and national pandemic statistics characterizing factors such as testing rates, case prevalence, and hospitalizations. As of June 2021, the NYT visualization consists of the three main views illustrated in Figure 4.3. The view in Figure 4.1 is a bar chart listing the daily new case counts since the start of the pandemic—the bar chart is overlaid with a line chart listing the 7-day rolling average. A tool tip shows the rolling average for a given day. Figure 4.2 shows the second view, a county-level map of the US,

[1] `https://www.nytimes.com/interactive/2021/us/covid-cases.html`

[2] https://github.com/nytimes/covid-19-data

[t]

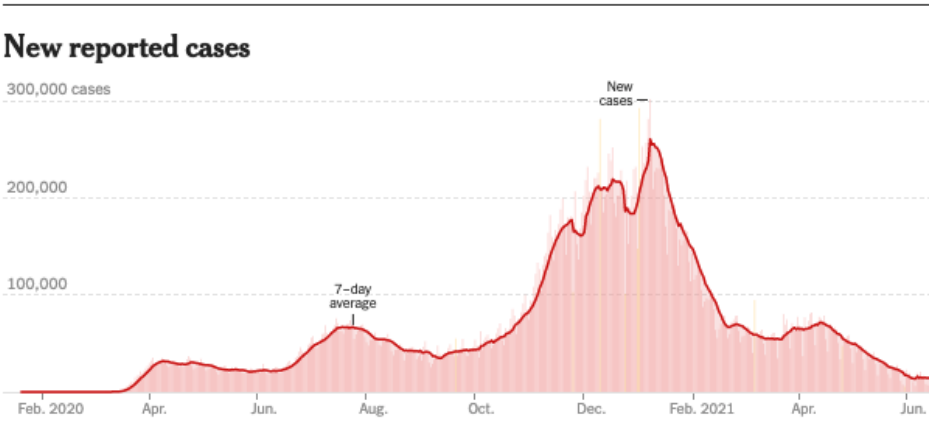


Figure 4.1: Daily reported COVID-19 cases in the USA (bars) and 7-day rolling average (line).

[t]

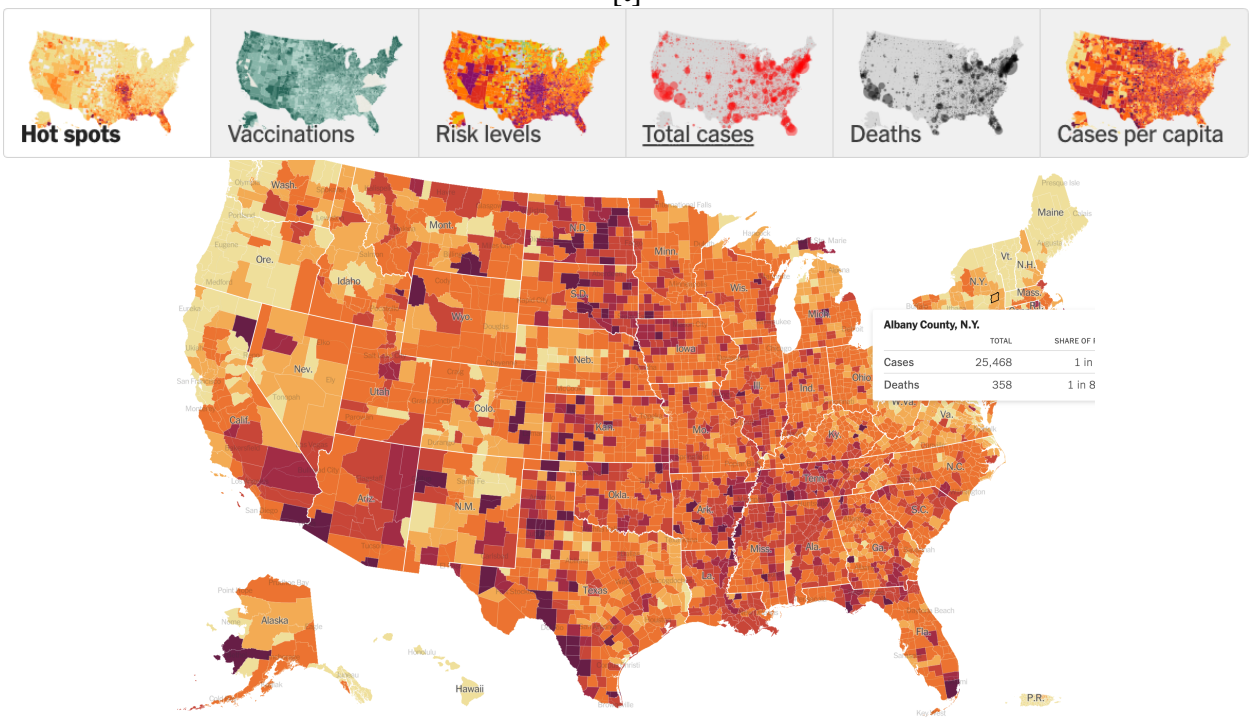


Figure 4.2: County-level map renders one of sive COVID-19 metrics, chosen via the panel above the map. Hovering over a county shows detailed statistics for the metric.

Figure 4.3: July 2021 New York Times US COVID visualization.

where a user can color code the counties by different statistics, e.g., risk levels, total cases, deaths, vaccinations.

Changes Over Time: Unsurprisingly, both the interface design and the underlying system implementation of the NYT visualization changed over the course of the pandemic. Initially, the available data was small, and directly loaded and processed by the user's browser. Since the available data was sparse, the bar chart only reported daily case counts, and the map reported statistics at the state granularity.

As more COVID-19 data was collected, the bar chart began reporting 7-day rolling averages and the map also offered statistics at the county granularity—users could then select a state or county to interactively update the bar chart. Further, the interface was steadily updated to incorporate additional statistics as they became reliably reported, e.g., hospitalization and vaccination counts.

Finally, as the dataset grew substantially over time, it needed to be pre-processed and served from a backend server to keep the interface responsive. The live NYT graphic uses the Svelte.js framework to maintain responsiveness, a library similar to React.js with the advantage of pre-computing many of the calculations performed that React.js performs in-browser. However, not all interactions were amenable to pre-processing. For instance, users expected the bar chart to update quickly when selecting a new county or state. Given the formidable cost of loading precomputed averages for every statistic and county pair, this approach could not be done at "interactive speeds." As a result, the interface developers opted to have the interaction for choosing a county result in a page reload, which users are accustomed to taking a longer amount of time.

Takeaways: This example illustrates several common properties associated with designing and maintaining interactive visualization interfaces: (1) the interface is redesigned regularly, either in response to data changes, user needs, or performance opportunities/limitations, e.g., selecting a county changed to a page reload rather than an interactive update to the bar chart;(2) data changes commonly require architectural changes, e.g., moving from an in-browser to a client-server model, and the set of pre-computed data changes based on interface redesigns (e.g., the map granularity)

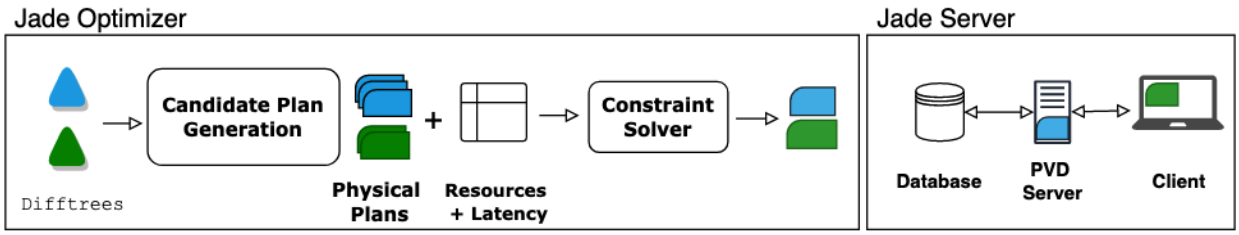


Figure 4.4: An overview of Jade.

and the presence of new statistics; (3) users implicitly expect different response times for different interaction types, e.g., page reload versus fluid interactions.

PVD tools such as Jade are meant to aid developers by decoupling the interface designer from the underlying architectural concerns. The next section describes how Jade takes as input a representation of the interface's data flows , called Difftrees, and uses it to optimize, recommend, and deploy a suitable execution architecture.

4.3 System Overview

Jade is a middleware that manages the optimization, deployment, and execution of interactive visualizations (Figure 4.4). In the offline design phase, Jade takes as input a representation of the interface's underlying data flows, the designer's responsiveness expectations for different interactions, and memory budgets on the client and server, and performs physical optimization to determine whath data structures to materialize, and an execution plan that spans the client and server. In the online interaction phase, the visualization responds to user interactions by calling the Jade client runtime to update the data flow state; the runtime re-runs the appropriate subplans, and passes updated query results to the visualization in order to update the interface. Below, we use the county-level heatmap metrics and and tool-tip interactions (Figure 4.2) to walk through both phases.

4.3.1 Design Phase

Interface Definition. Each heat map metric is a group-by aggregation query. For instance, the Hot Spots (left) and Cases per Capita (Right) maps are expressed as:

```
SELECT county,avg(cases)  SELECT county,sum(cases)
FROM us_covid_data FROM us_covid_data
GROUP BY county GROUP BY county
```

Hovering over a county renders a tooltip with additional statistics for the countys; the specific statistics depending on current metric. For instance, the Cases per Capita map displays the total cases and deaths:

```
SELECT sum(cases), sum(deaths)
FROM us_covid_data
WHERE county = "Albany"
```

Each set of queries is rooted in a common query structure, but vary in the **highlighted** portions—these are precisely the variations that `Jade` explicitly models in order to perform physical visualization design. To do so, we extend the SQL grammar and query operators with the notion of Choice Nodes that encode this variation. At a high level, choice nodes are a generalization of sargable parameters to arbitrary expressions, lists, and query fragments. We call query plans with choice nodes `Difftrees`.

Our syntax is inspired by SQL Server's parameterization syntax `[@NAME] TYPE-DEF` that defines the choice node and optionally assigns it the `name`. As in existing systems, a parameterized literal or column identifier specifies a name and type (e.g., `@cty str`), and can be bound to any type-compatible value. We further introduce `ANY` and `MULTI` types that respectively choose a single element or list of elements from a pre-defined set of choices. `ANY`$\{c_1, .., c_k\}$ can express any of its k children. `MULTI[op]{C}` can express a list containing zero or more bound copies of its argument `C`, concatenated by the operator `op`; `C` is almost always a `ANY` type. For instance, the set of tooltip and map queries described above can be expressed as:

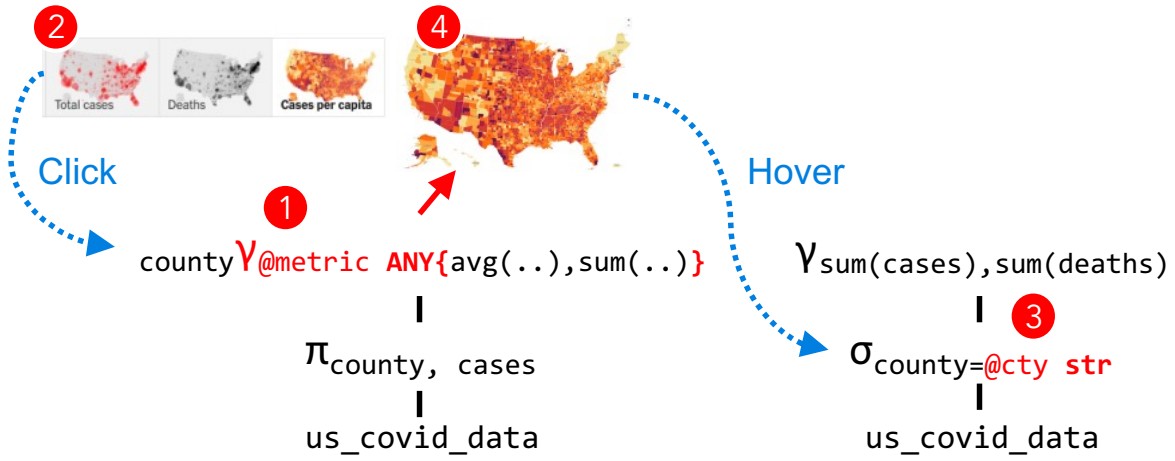


Figure 4.5: Difftrees for NYT Covid map visualization. Blue arrows denote interactions that bind to Choice Nodes in the Difftrees.

```
Q1:
SELECT sum(cases),
FROM us_covid_data
WHERE county=@cty str
GROUP BY county

Q2:
SELECT county, @metric ANY{avg(cases),sum(cases)}
FROM us_covid_data
GROUP BY county
```

The arguments to ANY and MULTI can be any expression or query statement (including subquery expressions), and they can be used in any place that expressions and query statements are allowed. Choice nodes with query statements as arguments are parsed similarly to the SQL UNION operator; otherwise, they are parsed as logical operators that take a list of expressions as arguments.

All possible combinations of choices in a Difftree generate a unique, syntactically valid SQL query. Maintaining syntactic validity is straightforward; a choice node type checks if each of

its individual choices would type check. §4.4 discusses choice nodes and their semantics, including correctness, in more detail.

We have implemented the NYT Covid visualization, as well as 8 additional real-world applications, using this syntax. In addition, recent work such as Precision Interfaces [CW20, CW21] have developed methods to generate `Difftrees` directly from example queries.

Specifying User Interactions. Choice nodes describe the variation that users interactions can specify. Once a developer registers a `Difftree` query with the system, she can bind the named choice nodes to values based on client interactions—a value choice can be bound to any type-consistent value, `ANY` can be bound to one of its arguments, and `MULTI` accepts a list of bindings that are each applied to its argument.

For instance, the following Javascript code registers the map and tooltip queries, updates the map visualization whenever the result of `Q2` changes. When the user clicks on a metric, the `onClick` handler extracts the index of the metric `idx`, and binds the choice node `@metric` to it; it further specifies that the interaction should be fast.

```
tooltipdt = Jade.register(Q1, "tooltipdt")
   mapdt = Jade.register(Q2, "mapdifftree")

mapdt.onData((table) => renderMap(table))

iact = Jade.registerIact({
   updates: [metric],
   latency: "fast"})

let onClick = (event) => {
  idx = getMetricIdx(event.x)
  iact({metric: idx})}
```

To put things together, Figure 4.5 illustrates the corresponding `Difftrees` for the above

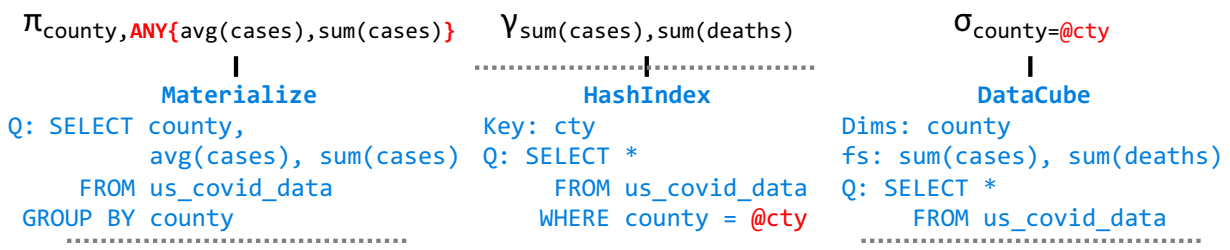
queries, along with the end-to-end interaction flow. (1) shows the `Difftree` for the different heatmap metrics, where the specific aggregation function is parameterized using an `ANY` operator. (2) clicking on one of the metrics binds `ANY` operator to a specific argument (e.g., `avg(cases)`), which will cause the query to re-run; its results are then re-rendered in the map visualization (4). Finally, when the user hovers over a county, its name is bound to the `@cty` choice node which re-runs the tool-tip query.

Physical Design.

The `Jade` physical designer is built on top of the Apache Calcite rule-based query optimization framework. The registered `Difftrees` are first parsed into an abstract syntax tree which is validated and then converted into a tree of logical operators. The designer takes as input these logical plans, interactions, latency expectations, and memory resources on the client and server. It then proposes data structures to materialize, along with an execution plan over these data structures, that will meet the latency expectations while meeting the memory budget. The data structures are chosen from an extensible library that includes hash indexes, data cubes, and spatial indexes (a full list is in Table 4.1).

`Jade` models each data structure in its library as an optimizer rule. It defines a tree pattern that matches fragments of a `Difftree`, and replaces the matched subtree with an operator that represents the data structure. We define transformation rules for the choice nodes and add optimizer heuristics to preferentially rewrite the query plan to maximize the likelihood of a data structure match. Each data structure is a physical operator that provides cardinality and latency estimates that integrate into Calcite's cost estimator. The optimizer independently proposes multiple candidate execution plans for each `Diffplan`, and `Jade` then encodes the plans into an integer linear problem to choose the combination of execution plans that meet the developer's latency and resource constraints. The output specifies the client/server caching policies for each data structure, and a split execution plan for each `Difftree`.

Figure 4.6 illustrates three example execution plans. Figure 4.6(a) chooses to materialize the entire `Difftree` query containing all of the projected expressions; the view is small enough



(a) Map Difftree (b) Tooltip Difftree 1 (c) Tooltip Difftree 2

Figure 4.6: Possible split execution plans for the map and tooltip `Difftrees`. Blue text denotes data structures to materialize, red text denotes choice nodes. The dashed gray line denotes execution in the browser (above) and server (below).

that it is pushed to the browser. The user interaction then projects the desired metric from the materialized view. In contrast, there are two candidates for the tooltip `Difftree`. Figure 4.6(b) replaces the filter with a hash index that is stored on the server; when the user hovers over a county, it probes the hash table and sends the results to the browser, which then aggregates the data. The alternative plan Figure 4.6(c) computes a data cube that natively supports slicing on `county`, and pushes it to the browser.

In order to perform the design optimization within a reasonable amount of time (under 1 minute for 9 real world applications), our main insight is to re-frame physical design from a cost minimization problem to a constraint satisfaction problem—the designer is either able to meet the developer's latency expectations or not able to. To avoid the need for accurate cost estimation, we draw from the perceptual cognition literature [LH14, BHP97, CMS99, CMN83, New90], which defines three time scales of human cognition. Perceptual fusion, such as tracking an animation or directly manipulating an object, occurs at $\approx 100ms$; cognitive operations, such as clicking on a link or selecting an object, takes $\approx 1s$; more complex or multi-step tasks occur at longer time scales. Following this categorization, we discretize latency expectations into fast ($< 100ms$) and medium ($< 1s$), and translate those respectively into constant or linear (or sub-linear) cost estimates for a candidate execution plan. This allows the optimizer to aggressively reduce the search space to plans that avoid complex joins. We describe these details in §4.5.

4.3.2 Interaction Phase

When the user loads the visualization, the `Jade` runtime manages re-computation in response to user interactions. As described above, the developer writes code to bind choice nodes in the registered queries to concrete values based on user interactions. Since the optimizer may have moved, split, or merged choice nodes during optimization, `Jade` tracks these transformations and the runtime automatically routes the developer bindings to the appropriate physical data structures (§4.4.4). `Jade` then reruns the execution plans that contain the bound choice nodes.

The execution plans are split across the server and client, and `Jade` inserts data transfer operators at the split points, similar to the Exchange operator [Gra94]. Since data structures and relational operators may be implemented in different languages and be executed on the client or server, serialization costs between physical operators can be prohibitive. `Jade` requires that all operators consume and produce data in Apache Arrow format [McK19], which enables zero-copy between operators, and allows the client and server to share the same operator implementations (Javascript in our prototype).

4.4 Interface Specification

A `Difftree` generalizes a logical query plan with three types of Choice Nodes that represent degrees of freedom, and user interactions bind these choice nodes to specific values. When all relevant choice nodes have been bound, the `Difftree` reduces to a standard query plan. In this way, a `Difftree` Δ is a compact representation of all of its expressible queries. Below, we first give a brief, informal overview of the `Difftree` structure, followed by its formal semantics. We then describe the `Difftree`-specific transformation rules used in the optimizer and provide guarantees of its correctness.

4.4.1 The `Difftree` Structure

Queries written in the extended SQL syntax are parsed into ASTs, which are then validated to verify the query is syntactically correct. We provide validation (type-checking) rules for choice nodes that ensure syntactic correctness. Once validated, an AST is planned into a tree of operators and expressions, following the same process as regular queries. Trees that contain choice nodes, which may exist in either operators or their expression trees are called `Difftrees`.

As shown in the previous section, the `ANY{c1,..,cn}` node chooses one of its child nodes. For instance, binding j will return c_j. The `ANY` node is flexible as it can express optional clauses (one child is a subtree, the other is a default/null),

The `VAL{type}` node is similar to `ANY`. It is is parameterized by a type and can be bound to any literal with that type.

Finally, the `MULTI[`θ`,n,m]{ANY{c1,..,cn}}` node is used wherever lists appear in the SQL grammar (e.g., project lists, group-by lists, conjuction of predicates, list of sources). Its child is always an `ANY` node, and binds its child between n and m times (by default, 0 and ∞, respectively). θ is an optional separator, used to define conjunctions or disjunctions. For instance, the predicate `county = ''Alameda'' OR county = ''Contra Costa''` can be generated by the following:

```
MULTI[OR]{ANY{county = VAL{str}}}
```

In practice, `ANY` nodes in expression trees are rewritten so that they only appear at the operator plan level. `MULTI` operators, which may only appear in lists, only appear as the top-level item in a list or pushed as close to the top of the list as possible. For instance, the filter operator $\sigma_{\texttt{ANY\{a,b\}=VAL\{num\}}}$ would be rewritten as $\texttt{ANY}\{\sigma_{a=VAL\{num\}}, \sigma_{b=VAL\{num\}}\}$, and the projection operator $\pi_{\texttt{[a, ANY\{b,c\}]}}$ would be rewritten as $\pi_{\texttt{ANY\{[a,b], [a,c]\}}}$. This helps simplify the set of transform rules in the optimizer.

We say a `Difftree` type checks if every query $q \in \{\Delta\}$ type checks. Column parameterizations (`VAL`) type check if their specified literal type would type check. An `ANY` is typed as a union

type of each of its individual choices; likewise, a `MULTI` is typed as a `List<T>` where `T` is the union type of its choices.

Validation rules for choice nodes can be summarized as follows: For all SPJ queries, an `ANY` or `MULTI` type checks (and can be validated) if each of its individual choices type check. If a query includes grouping or aggregation, its AST is valid if one of two conditions are met: (1) All choices in a `SELECT` clause are aggregations, or (2) all non-aggregation choices are in a static (non-choice) grouping list. Finally, choice nodes in `FROM` clauses or subqueries must be `UNION` compatible (i.e. always produce the same schema).

4.4.2 Formal Semantics

A `Difftree` Δ represents the set of possible plans that the interface can express through all combinations of valid bindings of its choice nodes. We denote this set as $\{\Delta\}$. For instance, the set of plans for `ANY{a,ANY{b,c}}` would be $\{a, b, c\}$. In this section, we formally define all operators in a `Difftree` with respect to the set of the plans they encode.

ANY is a choice node. Its children are an ordered tuple of subtrees consisting of either ANY nodes or SQL operators:

$$\emptyset : \textbf{ANY}(c_1 \dots c_n)$$

ANY may have any arbitrary number of children; however, an instantiation of an ANY node requires choosing exactly one of its children. As ANY has no operator type, it is preceded by the empty signifier $\emptyset$. When it is clear from the context, we drop $\emptyset$ to simplify the notation.

An ANY node with children $(c_1 ... c_n)$ represents the set of subtrees denoted by:

$$\{ANY\} = \bigcup_{i}^{n} \{c_i\}$$

If n = 1, then $\{ANY\} = \{c_1\}$. Note that by the definition of 4.4.2.1 below, this means that $ANY(c_1) = c_1$.

MULTI is a choice node that appears where lists can appear in the SQL grammar.

$$\Theta : MULTI\langle\emptyset : \textbf{ANY}(c_1 \ldots c_n)\rangle$$

MULTI's only child is an ANY node and can be instantiated multiple times. Each instantiation is an item from its associated ANY's children $c_1 ... c_n$. Multiple choices by default generate a list, but a combiner for logical expressions can be specified with AND or OR.

SQL operators are denoted by the form

$$\Theta : \textbf{SQL}\langle\theta : \textbf{EXP}\rangle(c_1 ... c_n)$$

Θ is the relational algebra operator type (e.g., σ, γ). θ : **EXP** is the parameterizing expression tree for that operator type, defined below. $(c_1 ... c_n)$ is an ordered tuple of child nodes. For example, the node $\pi : SQL\langle[attr_1 ... attr_n]\rangle(R_1)$ denotes a SQL node that projects $[attr_1 ... attr_n]$ and has a single child, the relation R_1. Where irrelevant, expression nodes are often omitted; e.g. the above can be written $\pi : SQL(R_1)$. The set of trees that a non-choice (SQL) node represents is derived from its children. The full set of possible subtrees rooted at a SQL node is the crossproduct of the diffsets of each child:

$$\{\Theta : SQL(c_1, .., c_n\}\} = \{\Theta(s_1, \ldots, s_n)\} | s_i \in \{c_i\} \forall i \in [1, n]\}$$

SQL Expressions are similar to SQL operators:

$$\theta : EXP(c_1 \ldots c_n)$$

Unlike SQL operators, all expression trees are bound to a $\Theta : SQL$ operator which they parameterize. Expressions can be attributes, literals (including numerics, text, or lists), functions, or logical operators such as `AND, OR,` $<, =, >$, such as projection lists, filter expressions, aggregate functions etc. Choice nodes may appear in expression trees, in which case their children must all be expression nodes. The diffset is then derived in a similar manner as for SQL operators. For readability, child tuples are omitted below.

$$\{\Theta : SQL\langle ANY(\theta_1 : EXP \ldots \theta_n : EXP)\rangle\}$$

$$= \bigcup_{i=1}^{n} \{\Theta : SQL\langle\theta_i : EXP\rangle\}$$

Note that we define the diffset of expression `Difftrees` in such a way that they can always be rewritten as a `Difftree` of the operator they parameterize, as in the following:

$$\{ANY(\Theta : SQL\langle\theta_1 : EXP\rangle \ldots \Theta : SQL\langle\theta_n : EXP\rangle)\}$$

Consequently, expression and operator trees are treated identically in the following transformations.

4.4.2.1 Equality

Informally, two `Difftrees` rooted at nodes named D_1 and D_2 respectively are equivalent if their diffsets are the same. Formally,

$$D_1 = D_2 \leftrightarrow \{D_1\} \subseteq \{D_2\} \wedge \{D_2\} \subseteq \{D_1\}$$

. Note that this does not require D_1 and D_2 to have the same diffnode type.

We further say that two `Difftrees` are equivalent if their set of plans are equivalent $\Delta_1 = \Delta_2 \iff \{\Delta_1\} = \{\Delta_2\}$, and that `Difftree` Δ_1 subsumes Δ_2 if $\{D_2\} \subseteq \{D_1\}$.

4.4.3 Transformation Rules

Given an initial `Difftree` Δ, there are a large number of different `Difftrees` that are equivalent or that subsume Δ, but enable different combinations of physical optimizations. In addition, data structures define matching patterns as `Difftree` fragments. For both reasons, it is useful to restructure and move choice nodes during optimization. To this end, we define three choice node transformation rules. These rules are safe in that the resulting `Difftree` is equivalent to the input `Difftree`, as demonstrated by the accompanying proofs.

`Jade` records the sequence of transformation rules that have been applied, so that interaction

bindings written relative to the original `Difftree` can be correctly routed and translated to the choice nodes in the final transformed plan.

4.4.3.1 Partition

A partition of a diffnode $A = ANY(a_1, \ldots, a_n)$ splits its children into a proper partition. Let $\Psi : [1, n] \rightarrow [1, m]$ be a partitioning function that maps each child of A to one of $m \leq n$ partitions. We define the resulting partitions s_1 to s_m as ordered lists:

$$s_i = [a_j | j \in [1, n] \wedge \Psi(j) = i]$$

We now define the partition function as follows, where we use splat notation to"unwrap" each partition s_i as arguments in a new ANY node:

$$partition_\Psi(ANY(a_1, \ldots)) \rightarrow ANY(ANY(s_1...), \ldots, ANY(s_m...))$$

The partition of is valid if the following conditions for partitioning a set are met:

- $\forall s_i, s_i \neq \emptyset$
- $\cup\{s_i...s_m\} = \{a_i : a \in A\}$
- $\forall s_1, s_2, s_1 \cap s_2 = \emptyset$

Proof. We show that $partition_\Psi$ as defined above preserves equivalence with respect to the result-

ing diffsets.

$$\{ANY(ANY(s_1...), \ldots, ANY(s_m...))\} \tag{4.1}$$

$$= \bigcup_{i=1}^{m} \{ANY(s_i...)\} \tag{4.2}$$

$$= \bigcup_{i=1}^{m} \{\bigcup_{j=1}^{k} a_j | \Psi(j) = i\} \tag{4.3}$$

$$= \bigcup_{i=1}^{m} \bigcup_{\Psi(j)=i} \{a_j\}\} \tag{4.4}$$

$$= \bigcup_{k=1}^{n} \{a_k\} \tag{4.5}$$

$$= \{ANY(a_1, \ldots, a_n\} \tag{4.6}$$

Equations 2 and 3 follow from the definition of a diffset, equations 4 and 5 follow from the definition of the partition function Ψ, equation 6 follows again from the definition of a diffset. □

4.4.3.2 Push

The push operation requires that the input subtree is rooted at a SQL-type node that has at least one child of type ANY:

$$P = \Theta : SQL(p_1..p_{k-1}, ANY(c_1..c_m), p_{k+1}...p_n)$$

The push operation "pushes" the diffnode one level higher through the parent. Let Q be the output node after the transformation; then,

$$P \rightarrow \emptyset : ANY(q_1 \ldots q_m)$$

where

$$q_i = \Theta : SQL(p_1...p_{k-1}, c_i, p_{k+1}...p_n)$$

We show that the push operation above preserves the equivalence {P} = {Q}.

Proof.

$$P(p_1 \dots p_{k-1}, ANY(c_1 \dots c_m), p_{k+1} \dots p_n) \tag{4.7}$$

$$= \{P(s_1 \dots s_n) : s_1 \in \{p_k\} \forall i \neq k, s_k \in \{ANY(c_1 \dots c_m\}\} \tag{4.8}$$

$$= \{P(s_1 \dots s_n) : s_1 \in \{p_k\} \forall i \neq k, s_k \in \bigcup_{i=1}^{m} \{c_i\}\} \tag{4.9}$$

$$= \{P(s_1 \dots s_n) : s_1 \in \{p_k\} \forall i \neq k, s_k = \{c_1\}\} \tag{4.10}$$

$$\dots \cup \{P(s_1 \dots s_n) : s_1 \in \{p_k\} \forall i \neq k, s_k = \{c_m\}\} \tag{4.11}$$

$$= \bigcup_{i=1}^{m} \{P(s_1 \dots s_n) : s_1 \in \{p_k\} \forall i \neq k, s_k = \{c_i\}\} \tag{4.12}$$

Equation 7 is the definition of the diffset of P. Equation 8 follows from the definition of the push operation and Equation 9 follows from the definition of an ANY diffset. Equations 10 (and 11, should be on one line) is a distribution of a union and Equation 12 is the definition a diffset. □

4.4.3.3 Merge

Finally, we allow a subtree rooted at diffnode to "merge" the children of a child diffnode into its own children, eliminating the merged child diffnode in the process:

$$ANY(c_1 ... c_{k-1}, ANY(d_1 ... d_m), c_{k+1} ... c_n) \rightarrow$$

$$ANY(c_1 .. c_{k-1}, d_1 ... d_m, c_{k+1} ... c_n)$$

Proof.

$$\bigcup_{i=1}^{n}\{c_i\} \tag{4.13}$$

$$= \bigcup_{i=1}^{k-1}\{c_i\} \cup \{c_k\} \cup \bigcup_{i=k+1}^{n}\{c_i\} \tag{4.14}$$

$$= \bigcup_{i=1}^{k-1}\{c_i\} \cup \bigcup_{j=1}^{m}\{d_j\} \cup \bigcup_{i=k+1}^{n}\{c_i\} \tag{4.15}$$

$$= \bigcup_{i=1}^{k-1}\{c_i\} \cup \{\{d_1\}\ldots\{d_m\}\} \cup \bigcup_{i=k+1}^{n}\{c_i\} \tag{4.16}$$

Equation 13 is the definition of a diffset. Equation 14 follows from the definition of a union. Equation 15 follows from the definition of a diffset and Equation 16 is the expansion of a union.

□

4.4.4 Semantic Rerouting

As diffnodes may be transformed with any of the three above operations which create, delete, and reassign diffnodes, the global mapping and input relations must all be rerouted accordingly. This process is detailed for each transformation type below. **Push** No new nodes are created or deleted during a push transformation, so no re-routing is needed. **Partition** A partition transformation creates m new choice nodes and reassigns the children of the transformed choice node. All of the newly choice nodes in the subtree also need a routing list.

Recall that the partition function creates ordered list:

$$s_i = [a_j | j \in [1, n] \wedge \Psi(j) = i]$$

To generate the routing table for the newly created node labeled @c, we define an inverse partition function unique to each partition that maps the original indexes to the partitioned indexes, where Ψ_i^{-1} corresponds to the diffnode with the ID i:

$$\Psi_1^{-1}(j) \rightarrow \begin{cases} \exists s_i[k] = a_j & k \\ else & -1 \end{cases}$$

Merge The merge operation removes a choice node from a `Difftree` by merging its children into its parents children.

For the parent diffnode @p with n children whose kth child @c merges its m children into @p, we define a piecewise routing function for the merge operation:

$$merge(@p, @c) = \begin{cases} @p \in [1, k-1] \rightarrow [1, k-1] \\ @p = k \rightarrow @p + @c - 1 \\ @p \in [k+1, n] \rightarrow @p + m - 1 \end{cases}$$

4.5 Physical Optimization

During physical optimization, `Jade` selects an optimization plan for the application. An optimization plan consists of a set of physical plans for each `Difftree`, with physical operators for each data structure encoding their respective placement policies. As in physical database design [CN98, BC08, DPA11], optimization happens in two steps. First, candidate physical plans are generated for each individual `Difftree`, with each plan contributing a potential set of data structures. In the second step, the combination of candidate plans for each `Difftree` is enumerated; each combination represents a potential optimization plan. `Jade` encodes each potential optimization problem as an integer linear programming (ILP) problem, along with constraints regarding resource usage and latency.

There are two unique challenges compared to PDD. First, PVD must navigate the very large search space inherent to the number of potential data structures and potentially infinite number transformations that can be applied to the query graph. Second, in order to meet strict latency

Jade Data Structure Library		
Name	**Description**	**Source**
Generic Datacube	multi-dimensional aggregation	Database
kd-Bush	2-D spatial index	Jade
Prefix Sum Index	cumulative sum	Jade
Sorted Index	range search on columns	Jade
Hashcube	multi-dimensional COUNT	External
Nanocube	multi-dimensional aggregation	External
BitFilter Index	static boolean filters	Jade
R-tree	multi-dimensional spatial index	Jade
Key-Value Cache	precomputed storage	Jade

Table 4.1: A summary of data structures implemented by Jade.

bounds, PVD considers all optimizing data structures in a single plan as a unit, rather than adding data structures to minimize overall latency.

Below, we first give an overview of the physical optimization process and cost model followed by how new data structures are added to Jade and a description of the optimization process.

4.5.1 Data Structure Library

Different interactive visualizations benefit from a diverse set of optimizing data structures. Along with general data structures such as data cubes, indexes, materialized views and key-value stores, certain types of interactions can only achieve sufficiently interactive speeds by leveraging bespoke data structures such as hashcubes [PSS16] or r-tree indexes [TLW19]. Consequently, Jade supports an extensible library of data structures that the optimizer chooses from, with our prototype implementing the data structures listed in Table 4.1.

4.5.1.1 During Optimization

`Jade` models data structures as physical operators with expression trees. For example, a (simplified) datacube operator defined as having two expression trees (a grouping list and an aggregation list), and a single child relational operator might define the constructor `Datacube(groupList: MULTI, aggList: ANY, child: RelOp)`. Additionally, data structure constructors may use union types, e.g. `MULTI<List<T>>|List<T>` indicating that an expression tree may be either a choice node or a static list. Union types allow more flexibility in defining operators. For simplicity, we use singly typed constructors throughout our example but note of how union types are accommodated.

Pattern matching rules provide instructions to the optimizer as to how the data structure can be used in a plan. Utilizing pattern matching rules builds on the modularity-preserving approach espoused by e.g. the Volcano optimizer [GM93] and modern frameworks such as Catalyst [CB18], enabling compatibility in many physical optimizers and flexibility in how data structure rules are applied.

Pattern matching rules transform a subtree in the `Difftree` to use a data structure operator. Patterns identify a subtree by matching node types and parent-child relationships. Nodes in the matched subtree can be bound to variables, which can then in turn be used to construct expression trees and arguments to the data structure operator. For example, a pattern matching rule that replaces an aggregation with the simplified data cube in our example might look like:

```
Aggregate(groups: MULTI<_>, aggs: _, child: _)
   =>
      new Datacube(groups, aggs, child)
```

_ indicates an unspecified (e.g. wildcard) type. Note that while here we borrow a `Scala`-like syntax for conciseness and readability, rules in the prototype use a different syntax. Multiple patterns may map to the same data structure; however, it is the developers responsibility to ensure that the initialization of the data structure conforms to its constructors expected types.

4.5.1.2 During Interaction

`Jade` manages query execution during application runtime. To do so, `Jade` requires that each data structure provide two functions, `build` and `query`.

`build` provides instructions for how to physically instantiate a data structure. It takes as input a table-like binary (an Arrow table) for each of its children. Its output is the a binary containing the physical data structure built by its expression tree. As it may be called by `Jade` before interaction begins, choice nodes in the data structure do not provide a binding and instead expose their type information, including a list of choices for `ANY` and `MULTI`. In the data cube example, `build` should create a data cube whose dimensions are all columns in the `MULTI` choice list and whose measures are all aggregates in the `ANY` choice list. Data structures with union typed constructors should inspect their expression trees and write build instructions accordingly.

`query` takes as input the data structure output by `build`. `Jade` guarantees that it will only be called during when bindings are available for all choice nodes in the `Difftree`. The body of `query` may make calls to the system function `Jade.bind()` for any of its expression trees that are typed as choice nodes. In the running example, `Jade.bind(groups)` will bind the `MULTI` choice node with a valid selection based on the application state. If `Jade.bind()` is incorrectly called on a non-choice node, it will simply return the static value, making it safe to use on union-typed expression trees. The output of `query` is a table-like binary (also an Arrow file) containing the results of the executed query.

4.5.2 Cost estimation

Data structure operators must also provide functions that estimate the latency and size of the data structure for use during optimization. To aid in calculating these metrics, data structures may access the global catalogue for table schemas and cardinality metadata.

Latency (cost) estimation is a generally difficult problem, and `Jade` takes a simplified approach, relying on two metrics to estimate latency. Latency calculations take into account data

transfer time and execution time. Data transfer depends on placement; if a data structure is co-located with the application, then its transfer time is zero. Otherwise, it varies with output cardinality (the amount of data transferred) and bandwidth speeds; `Jade` defaults to a transfer estimate of 10Mb/s but allows users to customize this value.

Execution time is estimated as a function of the number of rows processed during execution. Developers are therefore required to provide an estimate of `rowCount`. While developers are free to implement arbitrarily complex estimations, `Jade` provides pre-defined traits indicating that the data structure's processing speed is "fast", "medium" , or "slow", roughly corresponding to categorizations from human perception literature [LH14, CMN83]. These traits assume that "fast" data structures process a constant number of rows (i.e. ¡ 10 rows processed). "Medium" data structures are assumed to operate in linear time, and process rows amounting to the sum of their child operators `cardinality`. Typically, optimizing data structures are not "slow", but we nonetheless define this class as processing the cross product of their inputs.

`Jade` makes probabilistic estimates for choice nodes, assuming each choice has an equal probability of being chosen unless given an alternative distribution. In the absence of an explicit upper limit on the size of its list, a `MULTI` is estimated at its *maximum* size, which is the list produced by selecting each item from among its potential choices. Given a limit m, `MULTI` estimates its size and cost as the largest possible list with size m

Finally, developers should implement an estimation for the property `size`, indicate how much physical space the data structure requires when built. There is no default implementation for size; however, for the data structures used in the prototype, we found that data structures typically provide size functions in their associated literature that can be directly translated into the data structure operator.

With these three metrics (`size, rowCount`, and `cardinality`, `Jade` can recursively generate reasonably accurate estimates for both size and latency. `Jade` calculates `rowSize` for a given schema using catalogue metadata. Execution time for a single operator is then `rowCount * rowSize`, divided by the processing speed. Processing speed depends heavily on the backend

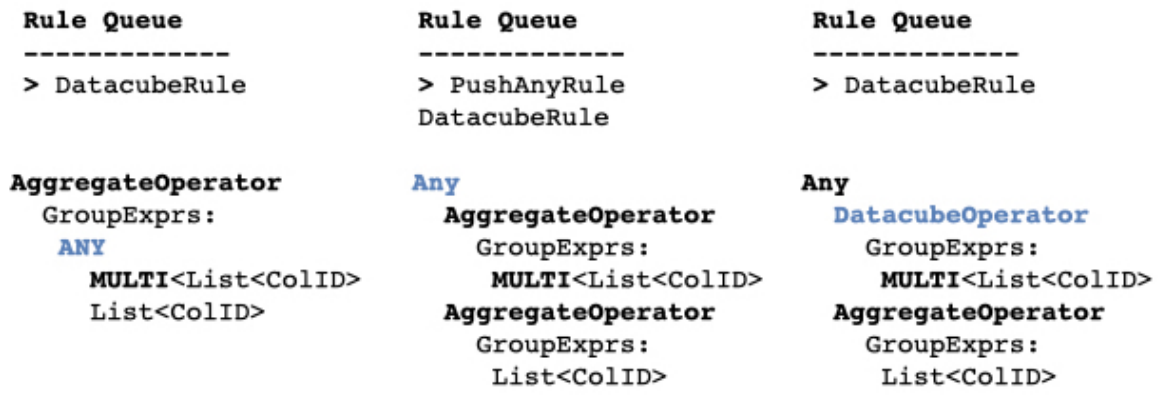


Figure 4.7: Conditional matching process on an input `Difftree`. In (1), the datacube transformation rule finds a conditional match, with the blue `ANY` interrupting a potential match. In (2), a `PushAnyRule` is pushed to the top of the stack, moving the `ANY` again highlighted in blue out of the way of the match. In (3) the datacube transformation rule fires again, this time successfully replacing a subtree.

architecture, but we empirically found that a single modern PC can process about 350kB/ms. . Similar calculations estimate data transfer (depending, again, on placement policies, discussed in Section 4.5.5) and resource consumption.

4.5.3 Per-`Difftree` Optimization

The first step in the optimization problem is generating candidate data structures for a single `Difftree`. Given a `Difftree` and a set of latency requirements, the problem at this step is to generate all potential physical plans meeting the latency requirement that are reachable by applying transformation rules. `Jade` assumes that at this point `Difftrees` have been canonicalized as described in Section 4.4.

Rules can be either the `Difftree` transformation rules described above or a data structure rule which replaces a logical subtree with a physical operator. Standard logical transformation rules otherwise supplied to the optimizer may also be included, but these rules typically do not

match choice nodes and are thus omitted from this discussion for brevity. In this section, we describe how Jade navigates the space of transformations and guides the search problem towards beneficial states.

At each step in the search problem, the Difftree Δ can take a transformation step t: $(\Delta) \rightarrow \Delta'$. Each node in the input Difftree is considered as a potential root of a matching subtree. If a potential root is found, pattern matching continues, recursively matching expression trees and then children until either the full pattern is matched, in which case the pattern is applied, or a mismatch is encountered, causing pattern matching to abort and begin at the next root. In an exhaustive search, Δ takes all transformation steps available, recursively applying transformations to the output of each transformation steps. As a baseline for Jade, we implement exhaustive search with the measures suggested by Volcano [GM93], including memoization, pruning of resource constraint violation, and cycle detection/elimination.

However, as we show in Section 4.6.3, we find that even optimized exhaustive search is either non-terminating or does not reliably find useful plans within a five-minute time limit on even small Difftrees. The primary reason for this is that Difftrees frequently need to be transformed to match the most useful data structure pattern, but depth-first exhaustive search either gets stuck in very deep, unhelpful branches while breadth-first exhaustive search meticulously explores many similar combinations of rule applications before moving on.

We use these observations to shape the heuristics employed by Jade to guide the search algorithm to useful transformations.

Heuristic Pruning. Based on the observation that Difftree transformations are useful only when used to move a Difftree towards a data structure pattern, Jade uses a technique called **conditional match pruning**. The heuristic pruning algorithm is based off a depth-first search similar to the exhaustive search described above, with two differences.

First, unlike exhaustive search, only data structure pattern rules are initially considered as part of a depth-first search. Second, when a mismatch is encountered, instead of aborting the current

rule, a conditional match search begins. Conditional matching allows mismatches if and only if the node in the input tree that caused a mismatch was a choice node (and can be "pushed" out of the subtree) or the direct child of a choice node (and can be "partitioned" out of the choice node). In either of these cases, the conditional match algorithm pushes the conditionally matched rule and the appropriate `Difftree` transformation rule onto the stack. The transformation rule then attempts to move the offending mismatching node, and the conditionally matched rules resumes an attempt to match the rest of the pattern.

For example, one application we encountered defines a default list of grouping dimensions for a dataset, but also optionally allows the user to select customizable groups. We represent this as an `Aggregate` operation whose `groups` expression tree is an `ANY` choice between a preset list or a `MULTI` operator (omitting irrelevant parts of the `Difftree`):

```
Aggregate(@g1 ANY{@g2 MULTI<List<ColID>>, List<ColID>}, ...)
```

This is a conditional match for the datacube transformation pattern, which matches a `MULTI` in a grouping list. As illustrated in Figure 4.7, if `@g1` is "pushed" out of the expression tree and into an operator, the datacube transformation rule will generate a candidate plan.

Conditional matching is employed as part of a memoized depth-first exploration. This means that *all* data structures that may match are explored independently, and transformation rules are applied recursively to the output of a successful transformation step. As we show in Figure 4.13 it is sufficient to drastically reduce the number of transformation steps and data structures explored, while still finding all of the unique plans that would be found in an exhaustive search.

4.5.4 Candidate Generation

Conditional matching is employed as part of a memoized depth-first exploration. This means that *all* data structures that may match are explored independently, and transformation rules are applied recursively to the output of a successful transformation step. As we show in Figure 4.13 it is sufficient to drastically reduce the number of transformation steps and data structures explored,

while still finding all of the unique plans that would be found in an exhaustive search.

After a successful data structure transformation step `t:` $(\Delta) \rightarrow \Delta'$, `Jade` further generates a plan reflecting three placement policies. In the first two placement policies, data structures storage and plan execution occur entirely on the client or entirely on the server. In the third placement policy, the execution plan is split at the first data structure(s) encountered from the root on each root-to-leaf path. This split divides the tree into a "top" half, containing the root (and no data structure operators), and a "bottom" half containing all data structure operators: $\Delta' \rightarrow (\Delta'_t, \Delta'_b)$.

In this plan, Δ'_b (and its data structures) is stored and executed on the server, with data structures sent to the client for execution of Δ'_t. This policy is particularly useful when Δ'_t is a subtree with only "fast" interactions and Δ'_b is a subtree with interactions that can tolerate longer latencies. All three placement policies are considered candidates in the configuration selection described next.

4.5.5 Cross-`Difftree` Optimization

Heuristic search yields a set of candidate physical plans for each view in an application. `Jade` must then optimize across all `Difftrees` in an application to find, if possible, a configuration that meets the resource and latency constraints. This problem can be formalized as a constrained integer linear programming (ILP) problem, as shown in physical database design [PA07], such as in AutoAdmin [BC08] and Cophy [DPA11]. `Jade` uses a problem formulation that takes a simplified approach as the latter; here we provide a brief description of the ILP formulation, highlighting modifications unique to `Jade`. For more detailed description of the general problem, we refer readers back to prior literature.

The major difference between our problem formulation and that of PDD is that unlike PDD, we do not consider a set of data structures and evaluate the cost of each `Difftree` depending on adding and removing data structures from the set. This is because we seek to meet *hard latency constraints*, where as PDD seeks to minimize the overall execution time of the workload. Therefore, we choose from a set of plans that provide guarantees about hard latency constraints, and do

not bother considering a data structure if it is not essential to a plan that meets those constraints. A set of data structures and a physical plan are a single unit in the configuration problem, allowing cost estimation to be inferred (and cached) directly in the plan.

ILP Formulation. We first consider an unconstrained version of the problem, where we seek to minimize overall latency without resource constraints. The inputs to the optimization problem are the resource budget for the client and server $\{B_c, B_s\}$ respectively, user-provided per-interaction latency constraints $L = \{l_1 ... l_n\}$, and a set of candidate plans for each `Difftree` in the interface. The goal is to select one candidate plan for each `Difftree` that satisfies the latency constraints for all interactions encoded in tree but that also fits in the resource budget.

Each candidate plan has an execution cost and a data transfer cost; its also contributes resource consumption from either the client, server, or both. We use β_{dk} to indicate the operator cost of `Difftree` d using the candidate physical plan k and use α_{dka} to indicate the data transfer cost if plan k for `Difftree` d uses placement policy a. The unconstrained problem (e.g ignoring resource and latency constraints) is then as follows.

Minimize:

$$\sum_{\substack{d \in D \\ k \in K_d}} \beta_{dk} x_{dk} + \sum_{\substack{d \in D \\ k \in K_d \\ a \in A_k}} x_{dk} y_{ka} \alpha_{dka}$$

for $x_{dk} \in [0, 1], y_{dka} \in 0, 1$

subject to:

$$\sum_{k \in [1, K_d]} x_{dk} = 1, \sum_{\substack{k \in [1, K_d] \\ a \in [1, A_k]}} y_{dka} = 1$$

x_{dk} controls the selection of a plan for `Difftree` d while y_{dka} controls the placement of each data structure in k. Note that k and a are both unique to each query, with K_d indicating the total set of plans for d and A_k indicating the set placement policies for plan k.

Adding Constraints. From the general problem, adding constraints is straightforward; for example, ensuring that `Jade` does not violate resource capabilities on the server and client are enforced by the constraint:

$$\sum_{\substack{k\in[1,K_d]\\a\in[1,A_k]}} y_{dka} \times size_s(a) \leq B_s$$

$$\sum_{\substack{k\in[1,K_d]\\a\in[1,A_k]}} y_{dka} \times size_c(a) \leq B_c$$

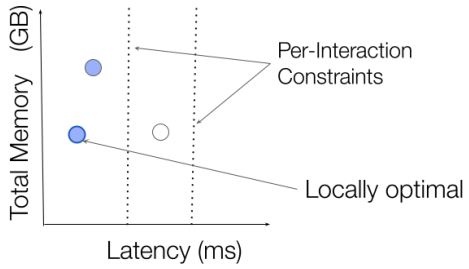
More importantly, the constraint mechanism is used to enforce *per-interaction latency bounds*. If we would only like to set a constraint against the *entire* `Difftree`, we can add a hard constraint to the cost term of d, e.g. $\beta_{dk}x_{dk} + \alpha_{dka}x_{dk}y_{dka} \leq 500ms$.

However, since a `Difftree` represents multiple interactions, we introduce another variable to enforce constraints per-interaction:

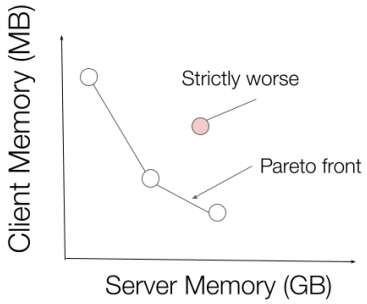
$\beta_{dki}x_{dk} + \alpha_{dkia}x_{dk}y_{dkia} \leq$ "$medium$"

This constraint specifies that the cost of the subtree containing the choice nodes used by interaction i in the `Difftree` d should operate within *medium* latency under the optimization plan for d; in effect, we treat the interaction as its own `Difftree`.Note that this only differs from the constraint against the entire tree if k uses a split-execution policy. Because `Jade` estimates cost recursively, per-interaction costs are obtainable directly from the physical plan itself.

Heuristic Approximation The above problem increases with complexity depending on the number of generated candidates, constraints, and the number of interactions. Solving it can potentially take minutes, as we show in evaluation. Thus, we devise a heuristic approach to its solution. The heuristic approach relies on two observations. First, as this is a constraint satisfaction and not minimization problem, as long as latency constraints are met for an individual interaction, there is no marginal benefit from speeding that interaction up further. Second, once we only consider plans that satisfy latency constraints, the only remaining constraints dictate the trade off in resource us-



(a) Only the blue-shaded plans satisfy latency constraints.



(b) Red-shaded plan is strictly worse.

Figure 4.8: Observations that allow for initial pruning of plans that either (a) fail to satisfy latency contraints or (b) are strictly worse in terms of resource consumption.

age between the client and server. Typically, reducing resource usage in one location causes an increase in usage on the other (accounting also for resource sharing between plans).

Together, these observations indicate that latency-satisfying plans form a Pareto optimization front with respect to resource usage. Noting that client resources are typically more constrained, our heuristic approach begins by minimizing client usage, selecting the Pareto-element for each `Difftree` that uses the least client resources. If client resources are overloaded, then there is no solution; however, if server resources are overloaded, then we explore the next solution by replacing the plan that maximizes the ratio $\frac{\Delta Server}{\Delta Client}$, e.g. the plan that adds the least additional resources to the client while removing the most from the server. This solution degrades to exhaustive exploration of all combinations of plans within their Pareto front; however, in our experiments, it finds an available solution if one exists within seconds.

4.6 Evaluation

In this section, we attempt to answer the following research questions about physical visualization design.

RQ1: How does `Jade` compare directly against commercial physical database design?

RQ2: How important is per-interaction granularity and visualization specific data structures in finding suitable physical designs?

RQ3: How long does optimization itself take?

To answer these we questions, we ported 9 visual applications found in news organizations, public galleries, and scientific publications to the `Jade` system (Table 4.2). For each application, we translated the charts and interactions into `Difftrees` and interaction mappings. We characterize three of the applications as simple because they have a single chart updated by one or two interactions. The rest are complex, as they mave multiple charts and interactions. Multiple charts are difficult because their interactions compete for resources. We characterize each applications complexity based on the number of views and interactions. Three of the nine applications are classified as simple, consisting of a single updated view and only one or two interactions. The remaining six applications we classify as complex, including the NYT Covid visualization shown earlier in Figure 4.3. These applications have more than one view that can be updated by the user (i.e. the bar chart and county map) and have multiple interactions that update different views (i.e. hovering over a bar, changing a metric, or selecting a county). Key to this distinction is that multiple views and interactions introduce competition for resources.

For each application, we downloaded their datasets and translated the charts, widgets, and interactions into the `Difftree` grammar. Several applications 'classified' as simple were intended

Interactive Application Benchmarks			
Name	**Complexity**	**Views**	**Interactions**
Cars [car]	Simple	Scatterplot of horsepower and miles per gallon	Click-drag selection retrieves cars in bounding box
Yemen Data Project [ydp]	Simple	Charts the number of missile attacks in Yemen	Retrieves details based on date and target
NPR Book Concierge [npr]	Simple	Displays book information and reviews	Filter by year and tags to retrieve matches
JHU Covid [jhu]	Complex	Global map of COVID statistics	Select country or metric, retrieve by date
NYT Vaccinations [nyta]	Complex	Map of vaccination rates in USA.	Explore by state, county, and age group.
NYT Covid [nytb]	Complex	Map of COVID statistics by state and county in the USA.	Set aggregation metric, explore by location
Flights [fli]	Complex	Bar and line graphs showing flight delay staistics.	Subaggregate by e.g. airline, airport, date
iCheck [ich]	Complex	Compares individual voting records against a reference group.	Set individual, reference group, date range
FairVis [CEH19]	Complex	Binary classification algorithm auditor	Set aggregate metrics and create arbitrary subgroups.

Table 4.2: Visualization applications used in `Jade` benchmark.

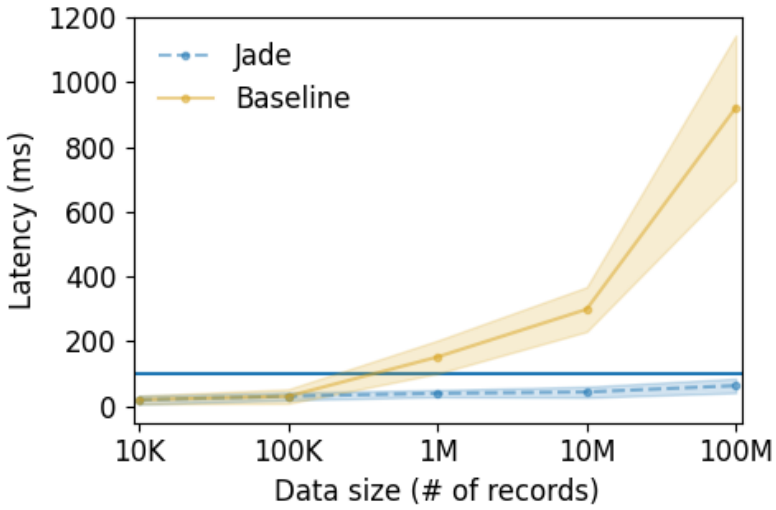
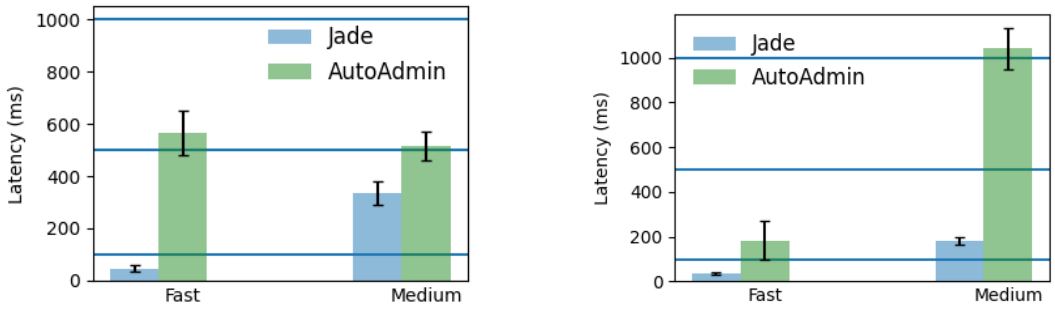


Figure 4.9: Scaling "simple" applications from 10k to 100M records. Reported values are the mean latency across all interactions. Filled area indicates 95th percentiles.

to run on very small datasets and, as part of their design, fetched all records matching a user selection. Because scaling such applications inevitably results in transferring untenable amounts of data, we modified these applications by adding a limit clause. Across all experiments, `Jade` is hosted on a 2015 15" MacBook Pro. We generate interactive input values by determining the domain of inputs for the interaction and using a random seed to select from that domain, weighted by the input distribution. As the baseline for comparison varies for each experiment, we describe them in the relevant section.

4.6.1 Scaling Single-Interaction Applications

In this experiment, we compare `Jade` to a commercial cloud database, Amazon Redshift, with automated physical design capabilities such as automated caching workload-sensitive selection of sort and distribution keys. We warm the Redshift (and auto-optimization selection) instance by discarding the first 15 trials and reporting over the next 10. To evaluate `Jade` in this context, we scale the input data for all simple applications from 10K records all the way to 100M records, and set the latency expectations for all interactions to "fast". Figure 4.9 reports the average latency of



(a) iCheck Application

(b) FairVis Application

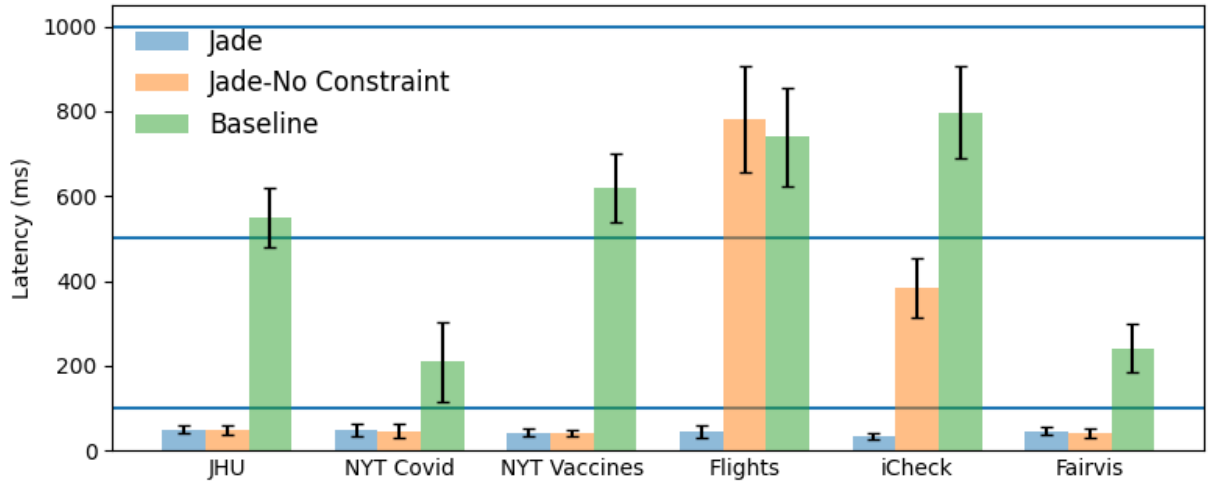
Figure 4.10: Comparison of physical design plans generated by `Jade` and the Database Tuning Advisor implementation of AutoAdmin in SQLServer on two representative applications.

all interactions across all three applications, along with 95th percentiles for each trial. Both the baseline and `Jade` are capable of meeting interactive latency expectations for data sizes up to 1M records. However, at or above 1M records the baseline begins to overshoot the desired latency. At 100M records, it can no longer guarantee even subsecond response times; by contrast, `Jade`'s performance remains nearly constant.

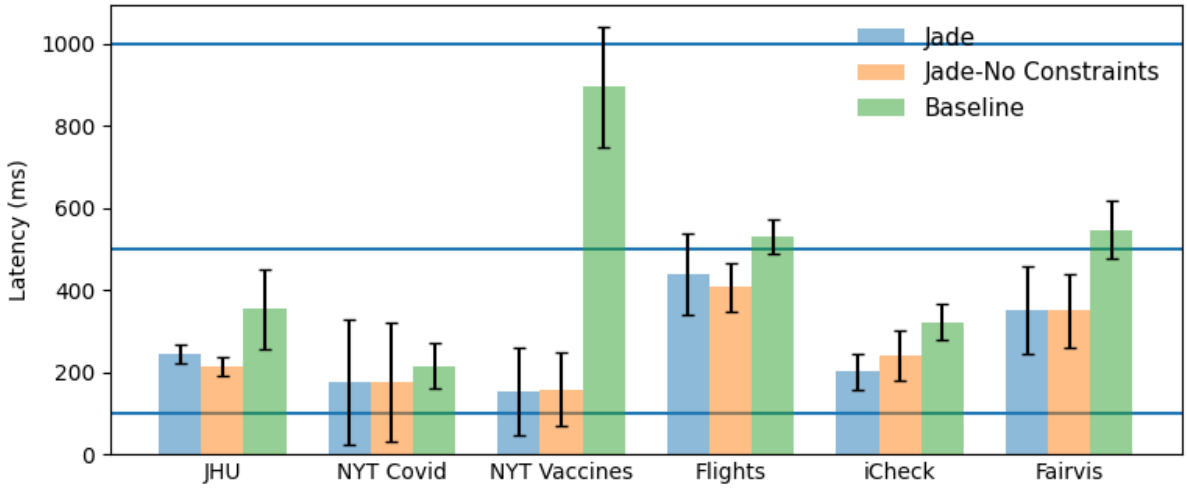
4.6.2 Optimizing Multi-Interaction Applications

Complex applications introduce both competition for resources and potentially conflicting latency constraints. In this set of experiments, we compare the ability of `Jade` to find suitable physical designs that satisfy latency constraints against commercial physical database design tools. First, we compare directly to the commercial implementation of AutoAdmin, SQLServer's Database Tuning Adviser. To retrieve recommendations from AutoAdmin, we loaded a database with the application data scaled to 10M records and provided the database tuning advisor with a sample 100 query workload, derived from interacting with the interface. The recommended indexes and views were then instantiated. Figure 4.10 shows the results in two representative applications.

The first application, iCheck represents an ideal candidate for optimization by AutoAdmin, as it is most amenable to support from the indexes and views supported by AutoAdmin, while the



(a) Latencies for interactions in the "fast" perceptual category.



(b) Latencies for interactions in the "medium" perceptual category.

Figure 4.11: `Jade` is the latency of the execution plan generated when the optimizer included a constraint reflecting the perceptual requirement. `Jade`-NC is the latency when the optimizer attempted to minimize overall latency (i.e. no constraint.) Dataset size is 10M records. Error bars indicate 5th to 95th percentiles.

latter, FairVis, is characterized by heavy re-grouping and aggregation. Even with a candidate rated highly amenable to standard database optimizations, the AutoAdmin baseline massively underperformed expectations. Neither application could achieve the desired interaction latencies using the AutoAdmin configuration.

Because of the sizable performance difference, we instead compare the remaining applications to the auto-optimized Redshift cluster. As an additional baseline, we also ran `Jade` on versions of the applications without setting any interaction latency constraints (`PVD-NC`), effectively treating it as a workload latency minimization problem as in AutoAdmin (but with the larger pool of potential data structures than actual AutoAdmin), rather than a constraint satisfaction problem. Figure 4.11 shows the median latencies across all interactions for the `fast` and `medium` perceptual categories per application.

As expected, Redshift is unable to achieve sub-100ms latencies, in part because of the need to issue, execute, and serialize data. Moreover, it tends to maintain a consistent performance across *all* perceptual categories, rather than optimizing for individual latency constraints. `Jade`-NC fares comparatively to unmodified `Jade` except in two cases, iCheck and FairVis. At first, it appears that without constraints, `Jade`-NC selects overall *worse* plans. Upon inspection, we realize that in the absence of latency constraints, `Jade`-NC optimizes for interactions that generate the *most* queries. Since interactions in the "medium" perceptual category generate many more queries in both applications displaying the discrepancy, `Jade`-NC selects plans that optimize for the heavier weight they contribute to the workload.

4.6.3 Optimization Time

To measure the efficiency of our optimization technique, we run two variations of the end-to-end optimization algorithm on all applications. Candidate generation is the time taken to generate candidates. An end-to-end optimization consists of candidate generation followed by either using an ILP solver or the heuristic described in Section 4.5.5 to find a solution from the candidates.

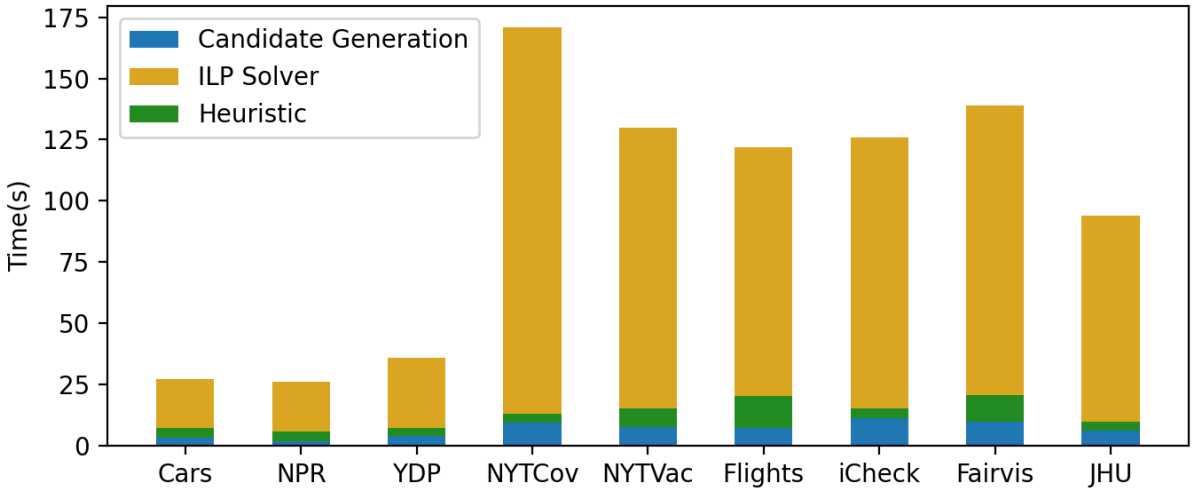


Figure 4.12: End-to-end optimization times for the nine applications used in the benchmark.

Both are reported in Figure 4.12

In addition, we demonstrate the necessity for using guided rule transformations to trim down the search space during candidate generation. We compare using our guided transformations to both exhaustive search and what we call optimized exhaustive search, which uses known cycle-breaking and symmetry reductions tactics to reduce the number of transformations. We randomly generate `Difftrees` with an increasing number of operators and choice nodes. Figure 4.13 shows the results as the number of steps taken, where a *step* is a `Difftree` generated by a transformation rule. Note that the y-axis is logarithmic. We cut off any search where over 5000 steps are taken.

`Jade` shows a clear linear trend. While optimized exhaustive search improves over exhaustive at small sizes, it eventually becomes indistinguishable. We note that although `Jade` takes a smaller number of steps, it ultimately finds the same number of unique *candidate plans*. Exhaustive and optimized exhaustive search do not lead to unique physical plans being generated; instead, they incur additional steps via spurious transformations of the logical tree.

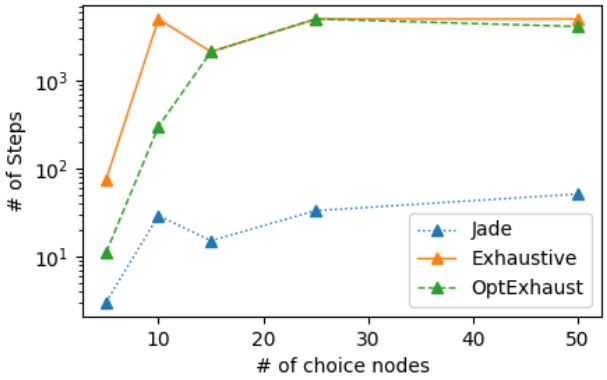


Figure 4.13: Number of steps taken in variations of per-`Difftree` candidate generation, scaled with increasing number of choice nodes. Note that y-axis is logarithmic.

4.6.4 Discussion

Direct comparisons to both commercial implementations of physical database design clearly indicate that such systems, while suitable for achieving reasonable performance at the workload level, are incapable of meeting the demands required by interactive applications, answering our first research question. Neither AutoAdmin or AmazonRedshift could reliably provide *fast* performance ($<$ 100ms) at moderate data sizes, beginning to exceed this threshold at around 1 million records on simple applications. On complex applications, comparing the latencies of interactions in the "fast" versus "medium" categories on baselines systems suggest that databases seek to maintain moderate performance across all queries in the workload, rather than the targeted allocation of resources needed in interactive applications.

Second, we investigate whether simply adding more data structures to existing physical design is sufficient to meet desired latency thresholds via `Jade-NC`, or whether interactive-level constraints are also necessary to achieve desired performance. `Jade-NC` behaves similarly to AutoAdmin, but introduces visualization specification optimizations. This modification demonstrates that without interaction-level constraints (that can target sub-regions of an individual query), `Jade` will, like AutoAdmin and Redshift, optimize for the weighted workload, resulting in undesirable (and sometimes overall worse) behavior. We can answer RQ2 by noting that visualization specific

data structures are necessary, but in some cases insufficient without per-interaction constraints.

Finally, in response to RQ3, Figure 4.12 illustrates that optimization time takes under three minutes using a full ILP solver, a time that can potentially be reduced to under 30 seconds with the use of the heuristic search described in Section 4.5.5. We deem this an acceptable time for a physical design tool that is intended to be run offline during development.

4.7 Related Work

Interface Design Tools: Many commercial and academic systems support interactive, SQL-based dashboards including Tableau, D3, and VizQL [STH02,Han06,HB10,BOH11]. Their architectures dynamically generate SQL query strings that are sent to a supporting database server. They do not support the integrated interface analysis and architectural optimizations proposed in this work.

Visualization-specific Optimizations Faster engines [PNV20,DCZ16,KN11], data cubes [LKS13, PSS16], precomputation, prefetching [BCS16], spatial indexes [TLW19] are all optimization frameworks proposed to guarantee interactive latencies for a subset of interactions. `Jade` analyzes the workload patterns each framework optimizes and systematically weigh a cost-benefit analysis of its use in a larger system. If the workload operations optimized by a given framework can be natively described in SQL, `Jade` can incorporate the framework into its optimziation plan. The main exceptions to this are binning [LJH13] and approximation [AMP13] techniques which are particularly difficult to describe in standard SQL; we view developing algebraic primitives for these operations as potential future work.

Parameterized Query Optimization Parameterized query optimization [BBD08, HS02, TK16] enumerates a set of query templates to be compiled by the database, with desirable query plans cached along the range of potential parameterizations. `Jade` differs in two key ways; first, it allows for safe parameterizations beyond simple literals extending into whole expressions and column groups. Second, `Jade` encodes program level context to guide data structure selection, while PQO is strictly concerned with query planning, not physical design.

Physical database design: Physical database design [FST88, MDA10, ZRL04, ADS10, CS13] solves a similar problem, identifying common access patterns in an arbitrary workload to build an appropriate set of indexes and views. However, as we discuss, there are several key differences between physical database design and physical visualization design; namely, the optimization goal (hard vs soft constraints), granularity of optimization (interaction vs workload or query [BC08]), architecture (database vs client-server), and the potential access patterns.

Self tuning and self driving databases: For instance, Pavlo et al [MVH18] model workloads as clusters of query templates, where a templates are considered equivalent if they access the same tables, use the same predicates, and return the same projections. In contrast, PVD explicitly specifies the query constructs that may be transformed, and thus does not require expensive, possibly incorrect query sampling and clustering steps.

4.8 Summary

We present `Jade`, a tool for physical visualization design. `Jade` builds on the principles of physical database design to introduce a novel language for describing interactive visualizations at the data level, a modular interface for introducing new optimizations, and search heuristics that leverage the properties of interactive visualizations to ensure tractable optimization times. Our prototype implementation demonstrates the utility of `Jade`, scaling and re-designing simple applications and managing resource competition among multiple interactions to maintain interactive latencies consistent with those required for perceptual interactivity.

REFERENCES

[Abh] AbhinavkumarL. "PageRank_InvertedIndex." `https://github.com/AbhinavkumarL/PageRank_InvertedIndex/`.

[AC18] Maaz Bin Safeer Ahmad and Alvin Cheung. "Automatically leveraging mapreduce frameworks for data-intensive applications." In *Proceedings of the 2018 International Conference on Management of Data*, pp. 1205–1220. ACM, 2018.

[ADS10] Ioannis Alagiannis, Debabrata Dash, Karl Schnaitter, Anastasia Ailamaki, and Neoklis Polyzotis. "An automated, yet interactive and portable DB designer." In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1183–1186, 2010.

[AMP13] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion 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*, pp. 29–42, 2013.

[Apa17] "Apache Hadoop." `http://hadoop.apache.org`, 2017.

[AX] Michał Świtakowski Alicja Luszczak, Michał Szafrański and Reynold Xin. "Databricks Cache Boosts Apache Spark Performance." `https://databricks.com/blog/2018/01/09/databricks-cache-boosts-apache-spark-performance.html`.

[AXL15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. "Spark sql: Relational data processing in spark." In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383–1394. ACM, 2015.

[BBD08] Pedro Bizarro, Nicolas Bruno, and David J DeWitt. "Progressive parametric query optimization." *IEEE Transactions on Knowledge and Data Engineering*, **21**(4):582–594, 2008.

[BBE15] MKABV Bittorf, Taras Bobrovytsky, CCACJ Erickson, Martin Grund Daniel Hecht, MJIJL Kuff, Dileep Kumar Alex Leblang, NLIPH Robinson, David Rorke Silvius Rus, JRDTS Wanderman, and Milne Michael Yoder. "Impala: A modern, open-source SQL engine for Hadoop." In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, 2015.

[BC08] Nicolas Bruno and Surajit Chaudhuri. "Constrained physical design tuning." *Proceedings of the VLDB Endowment*, **1**(1):4–15, 2008.

[BCS16] Leilani Battle, Remco Chang, and Michael Stonebraker. "Dynamic prefetching of data tiles for interactive visualization." In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1363–1375, 2016.

[BHP97] D. Ballard, M. Hayhoe, P. Pook, and R. Rao. "Deictic codes for the embodiment of cognition." *The Behavioral and brain sciences*, **20 4**:723–42; discussion 743–67, 1997.

[biy] biyingbin. "laobi-spark." `https://github.com/biyingbin/laobi-spark`.

[BOH11] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D^3 data-driven documents." *IEEE transactions on visualization and computer graphics*, **17**(12):2301–2309, 2011.

[car]

[CB18] Meenu Chawla and Vinita Baniwal. "Optimization in the catalyst optimizer of Spark SQL." *Turkish Journal of Electrical Engineering & Computer Sciences*, **26**(5):2489–2499, 2018.

[CCH16] Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, and Soudip Roy Chowdhury. *PigReuse: A Reuse-based Optimizer for Pig Latin*. PhD thesis, Inria Saclay, 2016.

[CDG08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. "Bigtable: A distributed storage system for structured data." *ACM Transactions on Computer Systems (TOCS)*, **26**(2):4, 2008.

[CEH19] Ángel Alexander Cabrera, Will Epperson, Fred Hohman, Minsuk Kahng, Jamie Morgenstern, and Duen Horng Chau. "FairVis: Visual analytics for discovering intersectional bias in machine learning." In *2019 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pp. 46–56. IEEE, 2019.

[CGD15a] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-centric Workflows." *Proc. VLDB Endow.*, **8**(12):1466–1477, August 2015.

[CGD15b] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and Stan Zdonik. "Tupleware: "Big" Data, Big Analytics, Small Clusters." 2015.

[Cha98] Surajit Chaudhuri. "An overview of query optimization in relational systems." In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 34–43. ACM, 1998.

[CMN83] S. Card, T. Moran, and A. Newell. "The psychology of human-computer interaction." 1983.

[CMS99] S. Card, J. Mackinlay, and B. Shneiderman. "Readings in information visualization - using vision to think." 1999.

[CN97] Surajit Chaudhuri and Vivek R Narasayya. "An efficient, cost-driven index selection tool for Microsoft SQL server." In *VLDB*, volume 97, pp. 146–155. Citeseer, 1997.

[CN98] Surajit Chaudhuri and Vivek R. Narasayya. "AutoAdmin 'What-if' Index Analysis Utility." In *SIGMOD*, 1998.

[CN07] Surajit Chaudhuri and Vivek Narasayya. "Self-tuning database systems: a decade of progress." In *Proceedings of the 33rd international conference on Very large data bases*, pp. 3–14, 2007.

[Cod02] Edgar F Codd. "A relational model of data for large shared data banks." In *Software pioneers*, pp. 263–294. Springer, 2002.

[cou] "CouchDB." `https://couchdb.apache.org/`.

[CR00] Chandra Chekuri and Anand Rajaraman. "Conjunctive query containment revisited." *Theoretical Computer Science*, **239**(2):211–229, 2000.

[CS13] Wei Cao and Dennis Shasha. "AppSleuth: a tool for database tuning at the application level." In *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 589–600, 2013.

[CW20] Yiru Chen and Eugene Wu. "Monte Carlo Tree Search for Generating Interactive Data Analysis Interfaces." *ArXiv*, **abs/2001.01902**, 2020.

[CW21] Yiru Chen and Eugene Wu. "PI2: Generating Visual Analysis Interfaces from Queries." In *ArXiv*, 2021.

[CWC17] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. "HoTTSQL: Proving Query Rewrites with Univalent SQL Semantics." In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pp. 510–524, New York, NY, USA, 2017. ACM.

[CY12] Rada Chirkova and Jun Yang. "Materialized Views." *Foundations and Trends in Databases*, **4**(4):295–405, 2012.

[Dat] Databricks. "Getting Started With Apache Spark on Databricks." `https://databricks.com/product/getting-started-guide/datasets`.

[DCZ16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. "The snowflake elastic data warehouse." In *Proceedings of the 2016 International Conference on Management of Data*, pp. 215–226, 2016.

[DDD04] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. "Automatic sql tuning in oracle 10g." In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 1098–1109, 2004.

[DN14] Christos Doulkeridis and Kjetil Norvåg. "A Survey of Large-scale Analytical Query Processing in MapReduce." *The VLDB Journal*, **23**(3):355–380, June 2014.

[DPA11] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. "Cophy: A scalable, portable, and interactive index advisor for large workloads." *arXiv preprint arXiv:1104.3214*, 2011.

[EA12] Iman Elghandour and Ashraf Aboulnaga. "ReStore: reusing results of MapReduce jobs." *Proceedings of the VLDB Endowment*, **5**(6):586–597, 2012.

[FBE09] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language." *Proc. LSDS-IR*, **8**, 2009.

[fli]

[FMC09] Marco Fiore, Francesco Mininni, Claudio Casetti, and C-F Chiasserini. "To cache or not to cache?" In *IEEE INFOCOM 2009*, pp. 235–243. IEEE, 2009.

[FST88] Schkolnick Finkelstein, Mario Schkolnick, and Paolo Tiberio. "Physical database design for relational databases." *ACM Transactions on Database Systems (TODS)*, **13**(1):91–128, 1988.

[GL01] Jonathan Goldstein and Per-Åke Larson. "Optimizing queries using materialized views: a practical, scalable solution." In *ACM SIGMOD Record*, volume 30, pp. 331–342. ACM, 2001.

[GMa] Ashish Gupta, Inderpal Singh Mumick, et al. "Maintenance of materialized views: Problems, techniques, and applications.".

[GMb] Himanshu Gupta and Inderpal Singh Mumick. "Selection of Views to Materialize in a Data Warehouse.".

[GM91] Goetz Graefe and William McKenna. "The Volcano optimizer generator." Technical report, COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE, 1991.

[GM93] Goetz Graefe and William J McKenna. "The volcano optimizer generator: Extensibility and efficient search." In *Proceedings of IEEE 9th International Conference on Data Engineering*, pp. 209–218. IEEE, 1993.

[gra] "GraphFrames Overview." `https://graphframes.github.io/graphframes/docs/_site/index.html`.

[Gra94] Goetz Graefe. "Volcano-an extensible and parallel query evaluation system." *IEEE Transactions on Knowledge and Data Engineering*, **6**(1):120–135, 1994.

[GRT10] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters." In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pp. 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[Hal01] Alon Y. Halevy. "Answering Queries Using Views: A Survey." *The VLDB Journal*, **10**(4):270–294, December 2001.

[Han06] Pat Hanrahan. "Vizql: a language for query, analysis and visualization." In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 721–721, 2006.

[HB10] Jeffrey Heer and Michael Bostock. "Declarative language design for interactive visualization." *IEEE Transactions on Visualization and Computer Graphics*, **16**(6):1149–1156, 2010.

[HPS12] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. "Opening the black boxes in data flow optimization." *Proceedings of the VLDB Endowment*, **5**(11):1256–1267, 2012.

[HS93] Joseph M Hellerstein and Michael Stonebraker. *Predicate migration: Optimizing queries with expensive predicates*, volume 22. ACM, 1993.

[HS02] Arvind Hulgeri and S Sudarshan. "Parametric query optimization for linear and piecewise linear cost functions." In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pp. 167–178. Elsevier, 2002.

[ich]

[IR95] Yannis E Ioannidis and Raghu Ramakrishnan. "Containment of conjunctive queries: Beyond relations as sets." *ACM Transactions on Database Systems (TODS)*, **20**(3):288–324, 1995.

[jhu]

[JKR18] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. "Selecting Subexpressions to Materialize at Datacenter Scale." *Proc. VLDB Endow.*, **11**(7):800–812, March 2018.

[JMH16] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. "SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment." In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 281–293. ACM, 2016.

[KA10] Kamal Kc and Kemafor Anyanwu. "Scheduling hadoop jobs to meet deadlines." In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 388–392. IEEE, 2010.

[KFM17] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. "ROBUS: Fair Cache Allocation for Data-parallel Workloads." In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pp. 219–234. ACM, 2017.

[Kie13] Alexander Kiel. "Datomic-a functional database." 2013.

[KM18] Michael Kinsey and Heather Miller. "Tour of Scala: Pattern Matching." `https://docs.scala-lang.org/tour/pattern-matching.html`, 2018.

[KN11] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In *2011 IEEE 27th International Conference on Data Engineering*, pp. 195–206. IEEE, 2011.

[LH14] Zhicheng Liu and Jeffrey Heer. "The effects of interactive latency on exploratory visual analysis." *IEEE transactions on visualization and computer graphics*, **20**(12):2122–2131, 2014.

[LJH13] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. "imMens: Real-time visual querying of big data." In *Computer Graphics Forum*, volume 32, pp. 421–430. Wiley Online Library, 2013.

[LK14] Jure Leskovec and Andrej Krevl. "SNAP Datasets: Stanford Large Network Dataset Collection." `http://snap.stanford.edu/data`, jun 2014.

[LKS13] Lauro Lins, James T Klosowski, and Carlos Scheidegger. "Nanocubes for real-time exploration of spatiotemporal datasets." *IEEE Transactions on Visualization and Computer Graphics*, **19**(12):2456–2465, 2013.

[LSH14] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigumus, Junichi Tatemura, Neoklis Polyzotis, and Michael J Carey. "Opportunistic physical design for big data analytics." In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 851–862. ACM, 2014.

[Mar06] Arun Marathe. "Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005." Technical report, 2006.

[MB] Imene Mami and Zohra Bellahsene. "A Survey of View Selection Methods.".

[McK11] Wes McKinney. "pandas: a foundational Python library for data analysis and statistics." *Python for High Performance and Scientific Computing*, **14**, 2011.

[McK19] W McKinney. "Introducing Apache Arrow Flight: A Framework for Fast Data Transport.", 2019.

[MDA10] Cristina Maier, Debabrata Dash, Ioannis Alagiannis, Anastasia Ailamaki, and Thomas Heinis. "Parinda: an interactive physical designer for postgresql." In *Proceedings of the 13th International Conference on Extending Database Technology*, pp. 701–704, 2010.

[men] mengxr. "spark-als." `https://github.com/mengxr/spark-als`.

[Mou18] Walaa Moustafa. "Transport: Towards Logical Independence Using Translatable Portable UDFs." `https://engineering.linkedin.com/blog/2018/11/using-translatable-portable-UDFs`, 2018.

[MRS01] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. "Materialized view selection and maintenance using multi-query optimization." In *ACM SIGMOD Record*, volume 30, pp. 307–318. ACM, 2001.

[MVH18] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. "Query-based workload forecasting for self-driving database management systems." In *Proceedings of the 2018 International Conference on Management of Data*, pp. 631–645, 2018.

[NBD12] Sandhya Narayan, Stuart Bailey, and Anand Daga. "Hadoop acceleration in an openflow-based cluster." In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pp. 535–538. IEEE, 2012.

[Neu11] Thomas Neumann. "Efficiently compiling efficient query plans for modern hardware." *Proceedings of the VLDB Endowment*, **4**(9):539–550, 2011.

[New90] A. Newell. "Unified Theories of Cognition." 1990.

[Nic] NicoViregan. "country-facts." `https://github.com/NicoViregan/country-facts/`.

[nod] nodesense. "gl-spark-scala." `https://github.com/nodesense/gl-spark-scala/b`.

[NPM10] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. "MRShare: sharing across multiple queries in MapReduce." *Proceedings of the VLDB Endowment*, **3**(1-2):494–505, 2010.

[npr]

[nyta]

[nytb]

[ORR15] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks." In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[ORS] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. "Pig Latin: A Not-so-foreign Language for Data Processing.".

[PA07] Stratos Papadomanolakis and Anastassia Ailamaki. "An integer linear programming approach to database design." In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pp. 442–449. IEEE, 2007.

[PMZ] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. "Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware.".

[PNV20] Panos Parchas, Yonatan Naamad, Peter Van Bouwel, Christos Faloutsos, and Michalis Petropoulos. "Fast and effective distribution-key recommendation for amazon redshift." *Proceedings of the VLDB Endowment*, **13**(12):2411–2423, 2020.

[PSS16] Cicero AL Pahins, Sean A Stephens, Carlos Scheidegger, and Joao LD Comba. "Hashedcubes: Simple, low memory, real-time visual exploration of big data." *IEEE transactions on visualization and computer graphics*, **23**(1):671–680, 2016.

[PTN18] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, Samuel Madden, and Matei Zaharia. "Evaluating End-to-end Optimization for Data Analytics Applications in Weld." *Proc. VLDB Endow.*, **11**(9):1002–1015, May 2018.

[RHH15] Astrid Rheinländer, Arvid Heise, Fabian Hueske, Ulf Leser, and Felix Naumann. "SOFA." *Inf. Syst.*, **52**(C):96–125, August 2015.

[RLG17] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. "Optimization of Complex Dataflows with User-Defined Functions." *ACM Comput. Surv.*, **50**(3):38:1–38:39, May 2017.

[RPE17] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. "Froid: Optimization of Imperative Programs in a Relational Database." *Proc. VLDB Endow.*, **11**(4):432–444, December 2017.

[RRS00] Prasan Roy, Krithi Ramamritham, S Seshadri, Pradeep Shenoy, and S Sudarshan. "Don't trash your intermediate results, cache'em." *arXiv preprint cs/0003005*, 2000.

[RSS00] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. "Efficient and extensible algorithms for multi query optimization." In *ACM SIGMOD Record*, volume 29, pp. 249–260. ACM, 2000.

[Ryu] Ryuka123. "kugou_music." `https://github.com/Ryuka123/kugou_music/`.

[sry] sryza. "aas." `https://github.com/sryza/aas/`.

[SSS16] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. "SourcererCC: Scaling Code Clone Detection to Big-code." In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 1157–1168, New York, NY, USA, 2016. ACM.

[STH02] Chris Stolte, Diane Tang, and Pat Hanrahan. "Polaris: A system for query, analysis, and visualization of multidimensional relational databases." *IEEE Transactions on Visualization and Computer Graphics*, **8**(1):52–65, 2002.

[TK16] Immanuel Trummer and Christoph Koch. "Multi-objective parametric query optimization." *ACM SIGMOD Record*, **45**(1):24–31, 2016.

[TLW19] Wenbo Tao, Xiaoyu Liu, Yedi Wang, Leilani Battle, Çağatay Demiralp, Remco Chang, and Michael Stonebraker. "Kyrix: Interactive pan/zoom visualizations at scale." In *Computer Graphics Forum*, volume 38, pp. 529–540. Wiley Online Library, 2019.

[tpc] "TPC Benchmarks." `http://www.tpc.org/default.asp`.

[TS] Dimitri Theodoratos and Timos K. Sellis. "Data Warehouse Configuration.".

[VCG10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. "Soot: A Java bytecode optimization framework." In *CASCON First Decade High Impact Papers*, pp. 214–224. IBM Corp., 2010.

[VH98] Raja Vallee-Rai and Laurie J Hendren. "Jimple: Simplifying Java bytecode for analyses and transformations." 1998.

[ydp]

[ZCD] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing.".

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing." In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZKJ08] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. "Improving MapReduce performance in heterogeneous environments." In *Osdi*, volume 8, p. 7, 2008.

[ZLE07] Jingren Zhou, Per-Ake Larson, and Hicham G Elmongui. "Lazy maintenance of materialized views." In *Proceedings of the 33rd international conference on Very large data bases*, pp. 231–242. VLDB Endowment, 2007.

[ZLF07] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. "Efficient Exploitation of Similar Subexpressions for Query Processing." In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pp. 533–544, New York, NY, USA, 2007. ACM.

[ZND01] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. "On supporting containment queries in relational database management systems." In *Acm Sigmod Record*, volume 30, pp. 425–436. ACM, 2001.

[ZRL04] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. "DB2 design advisor: Integrated automatic physical database design." In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 1087–1097, 2004.