Acorn: Aggressive Result Caching in Distributed Data Processing Frameworks

Lana Ramjit
UCLA

Matteo Interlandi
Microsoft

Eugene Wu
Columbia University

Ravi Netravali
UCLA

ABSTRACT

Result caching is crucial to the performance of data processing systems, but two trends complicate its use. First, immutable datasets make it difficult to efficiently employ powerful result caching techniques like predicate analysis, since predicate analysis typically requires optimized query plans but generating those plans can be costly with data immutability. Second, increased support for user-defined functions (UDFs), which are treated as black boxes by query engines, hinders aggressive result caching. This paper overcomes these problems by introducing 1) a judicious adaptation of predicate analysis on analyzed query plans that avoids unnecessary query optimization, and 2) a UDF translator that transparently compiles UDFs from general purpose languages into native equivalents. We then present Acorn, a concrete implementation of these techniques in Spark SQL that provides speedups of up to 5× across multiple benchmark and real Spark graph processing workloads.

KEYWORDS

result caching, user-defined functions, data analytics frameworks, materialized views, computation reuse

ACM Reference Format:
Lana Ramjit, Matteo Interlandi, Eugene Wu, and Ravi Netravali. 2019. Acorn: Aggressive Result Caching in Distributed Data Processing Frameworks. In *SoCC '19: ACM Symposium of Cloud Computing conference, Nov 20–23, 2019, Santa Cruz, CA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357223.3362702

1 INTRODUCTION

Recent years have witnessed significant efforts to improve the speed with which large-scale data processing frameworks like Apache Hadoop [4] and Spark [67] execute queries [35, 47, 53, 69]. A common technique used to accelerate data processing tasks is result (or view) caching [28, 41, 64, 72]. 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 [11, 15, 21, 26, 29, 33, 34, 38, 51]. 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 [8]. However, several major trends in recent data processing frameworks (e.g., Spark SQL [8]) 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 [68]. The reason is that, unlike databases which perform data updates in-place, modern analytics frameworks operate on immutable data [8, 9, 52]. 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 [68], 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 [7, 23, 59, 60], 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) [24]. The reason is that it is challenging to determine where in the query optimization pipeline (Figure 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' [19] 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 [8, 22, 43].

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

Unfortunately, query optimizers treat UDFs as black boxes, and must thus resort to exact-match caching. Recent work such as Froid [56] 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.

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 (§6.3). We make three main contributions.

Our first contribution adapts the extensive caching and predicate equivalence concepts from the database community [24, 39] 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 [55], Weld [54]), and can support the advanced language features described above that Froid [56] 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) (§5.3).

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 [3]) 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.

2 BACKGROUND

Using Spark SQL [8] 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 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 [4, 13, 25] share these steps, but Catalyst slightly differs in that stages 2-5 are lazily evaluated for efficient in-memory execution [68].

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 (§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 [14, 24, 32, 70].

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 [8].

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. However,

[2]Recent implementations[19] support automatic caching, where intermediate results are optimistically cached without user instruction.

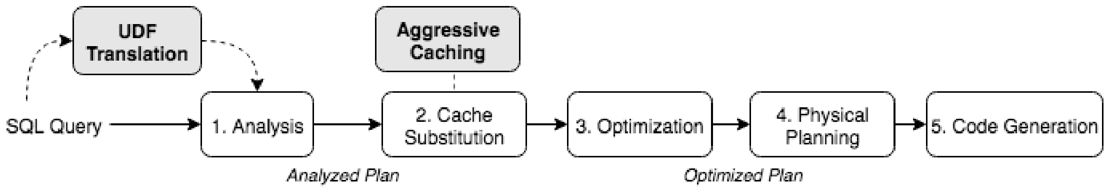


Figure 1: **Spark SQL's query planning pipeline. Grey boxes and dotted lines indicate Acorn's new components. Each stage generates or modifies a query plan and passes it down the pipeline; query plans relevant to Acorn are labeled.**

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 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`.

3.1 Need 1: Aggressive Identification of Result Caching Opportunities

Spark SQL's exact-match caching (§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:

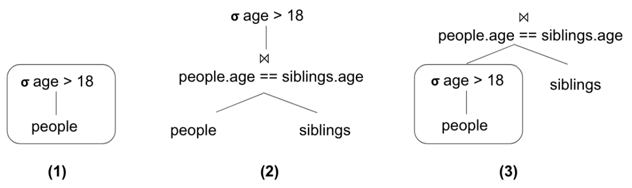


Figure 2: **(1) The (identical) analyzed and optimized plans for Query 3, (2) analyzed plan for Query 4, and (3) optimized plan for Query 4. Boxes show a cache opportunity: Query 3 can safely be used as Query 4's left join child.**

```
// 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 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.

3.2 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 (§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.

4 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.

4.1 Challenges and Approach

A natural and proven approach for detecting result caching opportunities (both exact match and subsumption relationships) is predicate analysis [14, 24, 32, 70]. 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 [27, 45, 71], we need not worry about stale (and inaccurate) cached data. However, the associated query optimization overheads (§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 §7.

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 §7 that these passes through the query optimizer result in significant resource and delay overheads [42, 48].

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 inexact query matches that predicate analysis identifies (which have proven to be significant in databases and early data analytics frameworks [11, 15, 21, 26, 33, 34, 38, 51]) 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.

4.2 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 [24]. 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 [24]. 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 §3). 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 §3). 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) [30].

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.

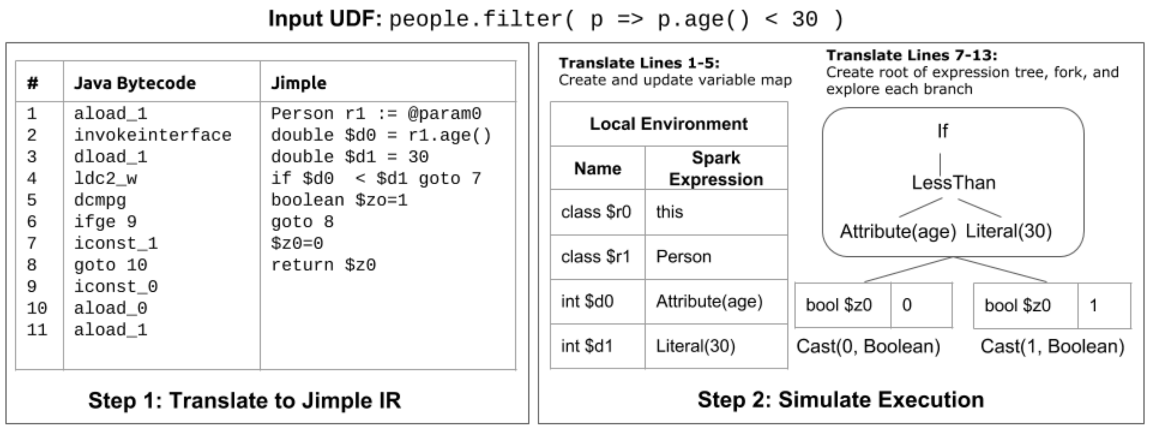
Correctness: We meet both conditions proven to preserve correctness when moving predicates [30]. First, our approach does not change the order of join operations. Second, we ensure that every root-to-leaf path in the query tree only refers to attributes produced by the join child. This follows from the 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.

5 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 §4.

5.1 Goals and Solution Overview

We have several goals and requirements for UDF translation that existing techniques do not meet (§8):


Input UDF: people.filter(p => p.age() < 30)
Translate Lines 1-5:
Create and update variable map
Translate Lines 7-13:
Create root of expression tree, fork, and explore each branch
If
LessThan
Attribute(age)
Literal(30)
bool $z0
0
bool $z0
1
Cast(0, Boolean)
Cast(1, Boolean)
Step 1: Translate to Jimple IR
Step 2: Simulate Execution

#	Java Bytecode	Jimple
1	`aload_1`	`Person r1 := @param0`
2	`invokeinterface`	`double $d0 = r1.age()`
3	`dload_1`	`double $d1 = 30`
4	`ldc2_w`	`if $d0  < $d1 goto 7`
5	`dcmpg`	`boolean $zo=1`
6	`ifge 9`	`goto 8`
7	`iconst_1`	`$z0=0`
8	`goto 10`	`return $z0`
9	`iconst_0`	
10	`aload_0`	
11	`aload_1`	

Local Environment	
Name	Spark Expression
class $r0	this
class $r1	Person
int $d0	Attribute(age)
int $d1	Literal(30)

Figure 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.**

- **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 (§8). UDF rewriting [54, 55] and annotation techniques [39] violate our transparency requirement, while program synthesis approaches [6] take minutes to run. Further, UDF compilers like Froid [56] 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 [37], 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 (§6) makes this entirely transparent to users.

5.2 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: `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`.

5.2.1 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 [66]. 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 [65], which adds minimal overheads to the overall translation process (§7). Figure 3 shows a comparison of our example lambda (written in Scala) with its Java bytecode and Jimple equivalents.

5.2.2 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 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 techniques 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), 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 [37]. 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). 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.

5.2.3 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 (§5.3), we revert to the original UDF path to ensure correctness; failed translation overheads are negligible (§7).

5.3 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 [56], 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.

6 ACORN

Acorn integrates our aggressive result caching optimizations (§4 and §5) into the query processing pipeline (Figure 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.

6.1 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 (§4) in 250 lines of code. If a cache opportunity is detected, this function rewrites and returns the query accordingly.

6.2 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 §4.

6.3 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 [42]), especially those that use immutable datasets (e.g., CouchDB [1], Datomic [36], BigTable [12]), 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 §4.2). 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 [43], DyradLINQ [22]) whose UDFs can be compiled to an IR (e.g., asm, .NET, LLVM).

7 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 §4 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 [56] cannot; and 5) overheads for Acorn's predicate analysis and UDF translation techniques are negligible.

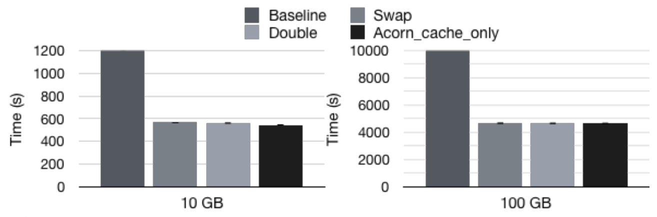


Figure 4: Evaluating Acorn on the SQL-only (no UDFs) TPC-DS workload. Bars represent median workload completion times (i.e., summing across all queries in the workload), with error bars spanning min to max.

7.1 Methodology

We evaluate Acorn on three main workloads:

TPC-DS v2.1: From this big data benchmark [3], 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 (§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 [3], 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 [56]. 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 [2]: 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 [40]: 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. [5, 10, 44, 49, 50, 61, 63]

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 §4: *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.

7.2 Aggressive Caching with Acorn

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

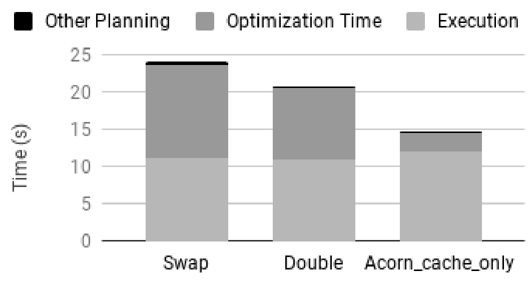


Figure 5: Breaking down Acorn's execution, compared to Swap and Double, on the 29 queries in the TPC-DS workload which automatically used some cached data. Results are for the 10 GB dataset, and times are the median of five runs.

Speedups: Figure 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 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 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.

Benefits of partial query optimization: Acorn, Swap, and Double identify the same caching opportunities in TPC-DS. However,

```
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 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.

Figure 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 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 (§7.1), the full optimization cost grows exponentially whereas Acorn's partial optimization grows linearly (Figure 7). The same trend holds for belief propagation, which we omit for space.

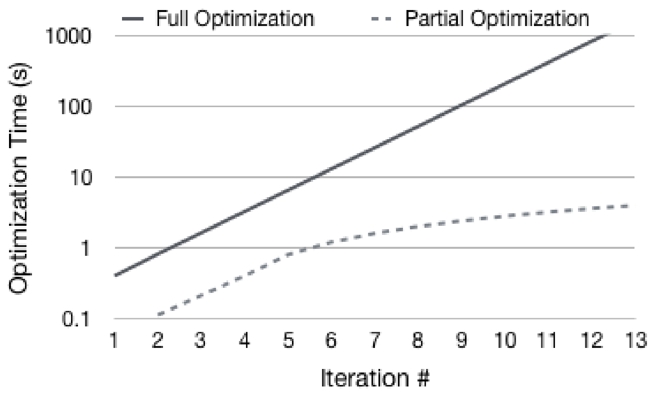


Figure 7: **Spark SQL's full optimization (Baseline) versus Acorn's partial optimization on the connected components algorithm. Note that the y-axis is logarithmic.**

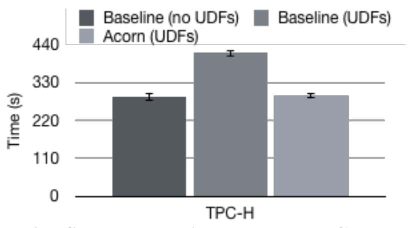


Figure 8: **Baseline Spark versus Acorn on the TPC-H workload (scaled to 10 GB), with and without UDFs.**

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.

7.3 Acorn's UDF Translation

We now evaluate Acorn's transparent UDF translation.

Cost of UDFs: Prior work in Froid [56] showed that UDFs incur a 100-10,000× slowdown compared to native relational operators. Thus, we first measure the overhead of using UDFs. Figure 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.

Scala

```
def q6cond(shipDate: Long,
    disc: Double, qty: Int) =
{
  val d1 = 757468799
  vale d2 = 31536000
  if ( shipDate < d1
    && shipdate >= d1 + d2
    && qty >= 24)
        return false
  val epsilon = .01
  val dec = .06
  var lower = dec - epsilon
  var upper= dec + epsilon
  return discount >= lower
    && discount <= upper
}
```

Native Spark

```
If(LessThan(shipDate,  757468799),
If(GreaterThanOrEqual(
shipdate, Add(757468799,
31536000)),
If(GreaterThanOrEqual(qty,
Literal(24)), If
(GreaterThanOrEqual(discount,
Subtract(Literal(.01),
Literal(.06))),
If(LessThanOrEqual(discount,
Add(Literal(.01), Literal(.06))),
true, false), false), false),
false, false)
```

```
 _.births > 100
```

```
GreaterThan(StructField("numBirth
s", IntegerType), Literal(100,
IntegerType)
```

```
line => {
  val fields =
line.split("[|]")
  IPRule(fields(2).toLong,
fields(3).toLong)}
```

```
with fields as Split(Col(0),
Literal("[|]", StringType)
    NewInstance(classOf[IPRule],
Cast(ElementAt(fields, Literal(2,
IntType), LongType),
Cast(ElementAt(fields, Literal(3,
IntType), LongType)))
```

Figure 9: **Original and Acorn-translated UDFs from real-world Spark workloads.**

Translating real UDFs: We extracted and ran the UDF translator on 30 Scala and Java UDFs from seven open-source repositories [5, 10, 44, 49, 50, 61, 63] 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; 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 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 §5.2 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 [56] *can* translate UDFs with such features.

The second and third UDFs contain examples of language features which Froid *does not* support. 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.

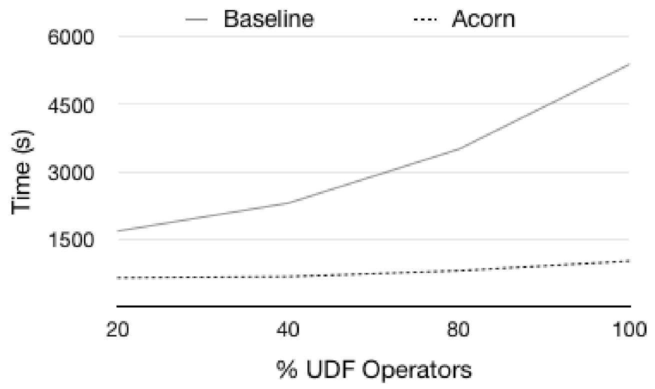


Figure 10: **Acorn vs baseline Spark SQL on the TPC-H workload (10 GB dataset), with varying fractions of query operators being replaced by UDFs.**

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.

7.4 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 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 (§7.1). 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×

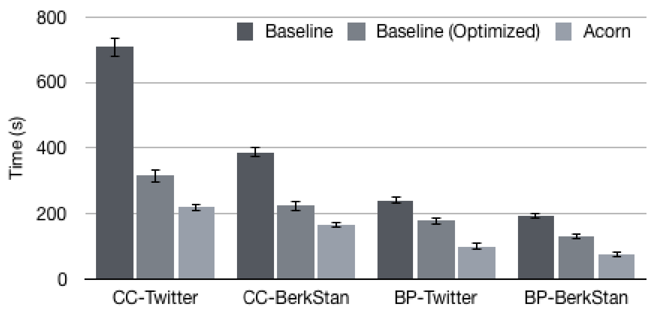


Figure 11: **Acorn vs baseline Spark SQL on two graph processing algorithms: connected components (CC) and belief propagation (BP).** `Baseline (Optimized)` **manually forces materialization of intermediate data structures. Experiments used snapshots of a Twitter follower graph (Twitter) and the Berkeley-Stanford web (BerkStan) [40].**

and 1.7× for connected components and belief propagation (Figure 11); Acorn's speedups are 2.3× and 3.2× over the baseline—all without any developer intervention.

8 RELATED WORK

Multi-query Optimization: Materialized views [24, 32, 70] and their maintenance [27, 45, 71] 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 [51] 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 [21] 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 [11] 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 [34] 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, heavily optimized languages for data analytics environments [46, 54, 55]. 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 [17, 18]. 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 [31, 57]. However, the derived annotations are not strong enough to detect subsumption relationships for aggressive caching [20, 58]. Other approaches have detected subsumption relationships with UDFs [39], 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 [56] 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 [16, 62] or program synthesis [6] 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 [6] 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.

9 CONCLUSION

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 paper 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 $2\times$–$5\times$ over Spark SQL. Though our implementation and results use Spark SQL, the underlying techniques generalize to other distributed data processing systems.

REFERENCES

[1] [n.d.]. CouchDB. https://couchdb.apache.org/.

[2] [n.d.]. GraphFrames Overview. https://graphframes.github.io/graphframes/docs/_site/index.html.

[3] [n.d.]. TPC Benchmarks. http://www.tpc.org/default.asp.

[4] 2017. Apache Hadoop. http://hadoop.apache.org.

[5] AbhinavkumarL. [n.d.]. PageRank_InvertedIndex. https://github.com/AbhinavkumarL/PageRank_InvertedIndex/.

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

[7] MichaÅĆ ÅŽwitakowski Alicja Luszczak, MichaÅĆ SzafraÅĎski and Reynold Xin. [n.d.]. Databricks Cache Boosts Apache Spark Performance. https://databricks.com/blog/2018/01/09/databricks-cache-boosts-apache-spark-performance.html.

[8] 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. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 1383–1394.

[9] 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. 2015. Impala: A modern, open-source SQL engine for Hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*.

[10] biyingbin. [n.d.]. laobi-spark. https://github.com/biyingbin/laobi-spark.

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

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

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

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

[15] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Foundations and Trends in Databases* 4, 4 (2012), 295–405. https://doi.org/10.1561/1900000020

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

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

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

[19] Databricks. [n.d.]. Getting Started With Apache Spark on Databricks. https://databricks.com/product/getting-started-guide/datasets.

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

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

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

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

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

[25] Goetz Graefe and William McKenna. 1991. *The Volcano optimizer generator*. Technical Report. COLORADO UNIV AT BOULDER DEPT OF COMPUTER SCIENCE.

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

[27] Ashish Gupta, Inderpal Singh Mumick, et al. [n.d.]. Maintenance of materialized views: Problems, techniques, and applications. ([n. d.]).

[28] Himanshu Gupta and Inderpal Singh Mumick. [n.d.]. Selection of Views to Materialize in a Data Warehouse. ([n. d.]).

[29] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (Dec. 2001), 270–294. https://doi.org/10.1007/s007780100054

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

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

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

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

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

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

[36] Alexander Kiel. 2013. Datomic-a functional database. (2013).

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

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

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

[40] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[41] Imene Mami and Zohra Bellahsene. [n.d.]. A Survey of View Selection Methods. ([n. d.]).

[42] Arun Marathe. 2006. *Batch Compilation, Recompilation, and Plan Caching Issues in SQL Server 2005*. Technical Report.

[43] Wes McKinney. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing* 14 (2011).

[44] mengxr. [n.d.]. spark-als. https://github.com/mengxr/spark-als.

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

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

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

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

[49] NicoViregan. [n.d.]. country-facts. https://github.com/NicoViregan/country-facts/.

[50] nodesense. [n.d.]. gl-spark-scala. https://github.com/nodesense/gl-spark-scala/b.

[51] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: sharing across

multiple queries in MapReduce. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 494–505.

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

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

[54] 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. 2018. Evaluating End-to-end Optimization for Data Analytics Applications in Weld. *Proc. VLDB Endow.* 11, 9 (May 2018), 1002–1015.

[55] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. [n.d.]. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. ([n. d.]).

[56] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 432–444. https://doi.org/10.1145/3164135.3164140

[57] Astrid Rheinländer, Arvid Heise, Fabian Hueske, Ulf Leser, and Felix Naumann. 2015. SOFA. *Inf. Syst.* 52, C (Aug. 2015), 96–125. https://doi.org/10.1016/j.is.2015.04.002

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

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

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

[61] Ryuka123. [n.d.]. kugou_music. https://github.com/Ryuka123/kugou_music/.

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

[63] sryza. [n.d.]. aas. https://github.com/sryza/aas/.

[64] Dimitri Theodoratos and Timos K. Sellis. [n.d.]. Data Warehouse Configuration.

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

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

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

[68] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

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

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

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

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