Practical Cross Program Memoization with KeyChain

Craig Mustard and Alexandra Fedorova
Department of Electrical and Computer Engineering
Universty of British Columbia

***Abstract*—Cross program memoization (CPM) reduces resource utilization and improves response times by enabling data processing systems to re-use previously computed results between programs. An under-explored requirement to implementing CPM in general purpose data processing systems like Apache Spark is computing identifiers for results of user-defined functions that are valid between programs while avoiding degrading system performance when sharing is not possible. In this paper we describe and evaluate a technique, called KeyChain, that computes keys for intermediate and final results of programs with user-defined functions. We use KeyChain to implement CPM in Apache Spark, and show that KeyChain's simple design means it can be easily added to relevant systems, incurs low runtime overheads, and enables heuristic detection of equivalent programs so that CPM can be added to more systems and useful results can be more widely re-used.**

I. Introduction

Data intensive programs can be long running and expensive to execute, so it can be quite beneficial to share previously computed intermediate and final results to speed up future computations. For example, if Alice and Bobbie are exploring the same data-set on the same infrastructure and Alice performs a join query that takes a long time, it would be ideal if the results of Alice's query are re-used when Bobbie runs the query. Similarly, if programs written by Alice and Bobbie perform the same pre-processing steps, they should also share their results. In general, it is desirable to share many previously computed results as widely as possible to improve performance and reduce resource usage. Since users of these systems may not even know if and when sharing can occur, the job of exploiting sharing opportunities falls to the execution system. **Cross Program Memoization (CPM) is our name for optimizations that enable programs run on shared infrastructure to re-use computation performed by other programs.**

In distributed data processing systems, cross program memoization (CPM) can yield significant gains. Nectar [1] reports a reduction in cluster execution time of up to 50% using CPM, SQLShare [2] estimates that 37% of their system's total execution time could be saved and BigSubs [3] realizes up to 40% machine time reduction on production clusters at Microsoft. Despite these successes, a core part of CPM implementations has remained under-explored in the literature: *low-overhead techniques to compute identifiers (keys) for the results of final and intermediate computations with user-defined functions (UDFs).* Such keys are needed to search for and look up results in a cache and to track metadata about previous computations, such as if a result *should* be cached next time it is computed.

Low-overhead key computation and CPM implementations are important because the potential to share results can vary widely between deployments. Ren et al [4] study a few deployments shared by researchers and find that different users share less than 1% of files, probably because each user was working on separate projects. Nectar's evaluation of automatic caching in production clusters shows some deployments experience less than 10% reduction in runtime, but others experience up to 50%. SQLShare [2] ran a long-term study on usage patterns of SQL-as-a-service and found that CPM could reduce total execution time by 37%, but most queries could either significantly benefit from sharing with a 90% reduction in execution time, or benefit very little, less than 10%.

Deployment-dependent variation in sharing potential presents an awkward choice to either require every deployment to somehow analyze benefits before enabling CPM, or leave it disabled and potentially miss out on beneficial opportunities. To break out of this dichotomy, **we need key computation techniques and, more generally, CPM implementations that have low overheads so that sharing can be enabled by default** in order to put *all* deployments in a position to exploit data sharing when it occurs. Despite this need, Nectar did not report overhead of their key generation or CPM implementation and Incoop [5], an incremental processing system, can increase an application's running time by 5-22%, a high price when no benefit is to be had[1].

Data processing systems like Apache Spark support UDFs so that users can perform arbitrary computations on data (§II-A). To increase sharing potential, a system would ideally detect if UDFs used in different programs are equivalent[2] so that it could more widely re-use results produced by UDFs. Systems that execute SQL are able to do this by analyzing query syntax [7, 8, 9, 10]. In contrast, UDFs are often written in the same general purpose language as the execution layer, so it is not immediately clear how the execution layer can detect UDF equivalence without implementing its own parser and compiler.

A. Contributions

This paper describes and evaluates a technique, called KeyChain, that computes keys for intermediate and final results of programs. We show this core component of CPM can be easily added to an existing system, incurs low runtime overheads,

[1]Incremental processing systems are usually applied on a per-application basis when the utility is known a-priori, so they are more tolerant to overheads on the initial execution.

[2]Since program equivalence is undecidable [6], the system would take a heuristic approach to keep overheads low.

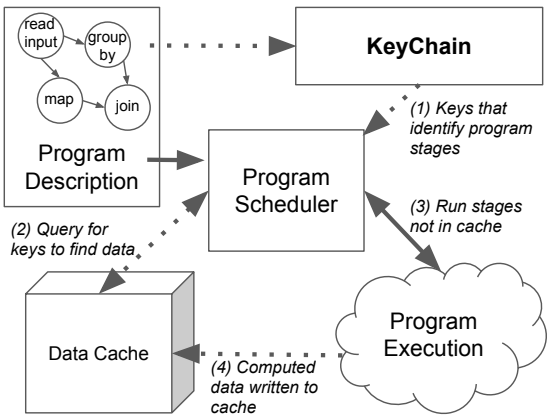

Fig. 1. Example execution layer augmented with KeyChain. Dotted lines show the generation and use of KeyChain keys in the system.

and enables heuristic detection of equivalent programs so data can be widely re-used. Figure 1 shows how KeyChain sits within a generic execution layer. KeyChain generates string-based keys that represent the intermediate and final results of data processing programs. Keys are valid across programs, so that different programs can re-use other program's results. KeyChain generates keys *before* the program is run by hashing together representations of input metadata, communication patterns, and the UDF bytecode. The execution layer of the data processing system can use these keys to find and store re-usable data in any storage system.

KeyChain addresses the challenge of computing a key that identifies the results of computations. We use KeyChain to implement a simple CPM in Apache Spark that enables sharing of previously cached results. We do not focus on determining *which* data to cache to achieve the best cost-benefit trade-offs as this important aspect has been covered extensively by prior work in memoization and automatic materialized view selection [1, 3, 7, 8, 9, 10, 11, 12] (§VI). KeyChain compliments this prior work by enabling systems to find more reusable data.

By presenting an evaluation of the feasibility and simplicity of KeyChain, we hope to encourage the wider adoption of CPM so that systems can share more data, reduce operating costs, and improve user experience. To this end, this paper makes the following novel contributions:

Simple Design. KeyChain is designed (§III) to be simple but effective so that CPM can be easily added to existing data processing systems that implement the DAG computational model (§II-A). We added KeyChain to Apache Spark's caching layer (§II-B) in under 2100 lines (§IV) that enables CPM for its existing in-memory cache (§V-B). The UDF hashing library and Spark implementation are open-source so that others can re-use and extend our work.

Low Overhead. KeyChain uses low overhead techniques to compute keys and the overheads grow with program size, *not* the data-set size. For example, prior work [1, 5] uses a hash of the input data as the input representation, requiring significant I/O costs to compute a key. Instead, KeyChain shows how to use input meta-data (e.g. a filename or query string) while still ensuring the computed keys reflect the most up-to-date input data. Our tests show KeyChain hashes UDFs in under 350ms (avg. 2ms) and our evaluation on TPC-DS [13, 14] shows no significant impact on query times when no sharing occurs. Low overhead means CPM can always be enabled because we pay almost nothing when sharing does not occur, but benefit significantly when sharing does occur (§V-D).

Evaluation of compiler-assisted UDF equivalence. Prior work [15, 16, 17] has shown it is possible to leverage existing compilers to heuristically detect functionally equivalent programs that are syntactically different. KeyChain's UDF hashing (§III-B) shows that we can use compiler-assisted equivalence detection to share data between some functionally equivalent UDFs, even if their syntax differs. To measure how well this effect works in Spark and motivate its use in other systems, this paper contributes a new benchmark suite to evaluate this property, called syntactic resilience, for different compilers (§V-C).

II. Background and Motivation

A. DAG Computation Model

Many popular distributed data processing systems represent computations as a directed acyclic graph (DAG), such as MapReduce [18] and Apache Spark [19]. In a DAG, each node represents an operation that produces data, and an edge from node x to y means that the output of x is used as input to y. DAGs enable an execution layer to understand data dependencies, which aids fault tolerance in the event of data loss, as well enabling optimization and scheduling decisions to be made before the program is executed.

For this paper, there are two important classes of DAG nodes: Input nodes and Transformation Nodes. *Input nodes* represent data read from some outside source, such as a network filesystem, database, or even randomly generated data. Input nodes have no incoming edges in the DAG, because they produce data from outside the computation model. *Transformation nodes* represent an operation applied to the node's input data to produce output data. Operations are defined as a combination of communication pattern (such as map, filter, reduce, and join) and a user defined function (UDF).

UDFs are an important part of the DAG computation model because they enable users to perform arbitrary computations on data, such as specialized transformations of input data objects, custom business logic, and development of new data mining and machine learning algorithms. To enable this flexibility, UDFs are written in a general purpose programming language, such as Java or Scala for Spark, and C/C++ for the original MapReduce [18]. UDFs provide such flexibility that Spark itself implements high level operations by combining UDFs with its lower-level operators such as building a join operation out of map and reduce.

As Spark and MapReduce do, this work assumes that UDFs are deterministic. Deterministic UDFs enables efficient fault tolerance mechanisms because any lost data can simply be re-computed using the information stored in the DAG. Neither Apache Spark or Hadoop MapReduce actually check UDFs

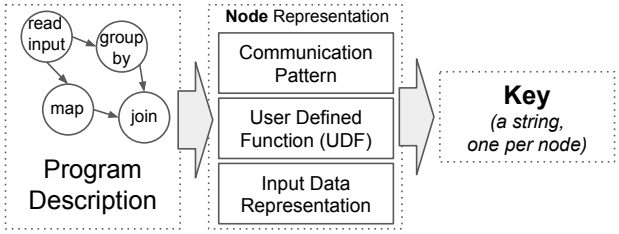

Fig. 2. How a key is generated by KeyChain.

for determinism, but we found this assumption to hold in all data processing programs we encountered.

B. Spark: A Motivating Example

Apache Spark is a distributed implementation of the DAG computation model[3]. In Spark, users use a simple API to compose together nodes. This DAG model also serves as the physical execution layer for higher level Spark APIs, such as Spark SQL and Dataframes [20]. In Spark, data in each node is not materialized until a user requests the data. At this time, the Spark execution layer will pipeline several transformations from related nodes together to produce an execution plan of stages to be executed.

To improve performance, authors of Spark programs can materialize and cache data of an important node to speed up subsequent computations[4]. As fig. 1 illustrates, when a node is to be materialized, the execution layer checks for any cached data that could be used, and if found, incorporates the data into the execution plan. *However, data and computation duplication occurs* because Spark identifies cached data using increasing integer IDs assigned to each DAG node. Since different Spark programs assign the same IDs for different data, it is not possible to share data between programs, even when they have equivalent nodes[5].

To share cached data more widely, we use KeyChain keys to identify cached data so that equivalent nodes will be assigned the same key. As we will show in the evaluation, this helps to share data when the *same* program has multiple equivalent nodes **or** when *different* programs have equivalent nodes.

III. KeyChain

KeyChain generates strings that represent a DAG node's output, which we refer to as keys. Figure 2 shows an overview of how KeyChain computes a key: string representations of the communication pattern (§III-A), UDF (§III-B), and input data (§III-C) are combined into strings that represent each node. The string that KeyChain generates is: KeyChain(node) = '`<communication pattern> <UDF> <input metadata>`'. This key can be used to query any storage system to find previously computed results that can speed up the computation.

[3]DAG nodes are called Resilient Distributed Datasets (RDDs) by the Spark authors

[4]Usually intermediate data is cached in memory, but it could also be serialized and written to an external storage system.

[5]Nodes are equivalent if they share the same input data, communication pattern, and UDF, thus producing the same output.

Keys for all DAG nodes are computed before the DAG is executed so that the cache can be checked for prior results. Prior work can also use this key to identify results that are useful to cache (§VI). Keys are computed starting from the input nodes, and the key of input nodes is used as the input data representation for transformation nodes (§III-C1). The next sections describe the representations KeyChain uses for input, communication patterns and UDFs.

A. Communication Pattern Representation

As described in §II-A, a communication pattern is an operation such as map, group by, or reduce, and this name can be used as the string representation. For instance, the '`map`' string describes a map operation that applies a UDF to each element. If the communication pattern takes any extra parameters that affect the output, such as the hash key for a shuffle operation, those parameters should be included in the string. (§III-C3).

B. UDF Hashing

A hash of the UDF is used as the UDF string in the key. Assuming the program is deterministic (§II-A), source code or any executable representation can be passed to the hash function. For the JVM (Java Virtual Machine), the best representation we found was the bytecode of the UDF because: (1) Overheads are low because bytecode is already produced during compilation, so KeyChain only needs to hash it. (2) Bytecode is less sensitive than program source to syntax variations like white-space, variable names, and comments, which means some functionally equivalent programs can share data, investigated further in §V-C. (3) Bytecode is accessible to programs running on the JVM (§IV). To further reduce runtime overhead in the future, it is possible for compiler to hash the bytecode at compile time. For instance, we developed an LLVM pass that hashes each function at compile time and stores the hash for lookup via function pointer at runtime.

The hash should capture the entire behaviour of the UDF, so our implementation hashes the UDF and all its callees. If a function is common across all programs in the system (such as the Java standard library) it is possible to skip hashing these functions and just include the fully qualified method name since no information about program behaviour would be lost.

C. Input Representation

We break computing input representations into two cases: input originating from outside the DAG, and from other nodes.

1) Internal Data: Chaining: To represent input from other nodes in the DAG, KeyChain uses the key of the node that supplies the input, called *chaining*. For example, if node x takes input from node y, then Key(x) = '`<communication pattern> <UDF> Key(`y`)`'. Since the key of input nodes is computed before other nodes, Key(y) will be known when Key(x) is computed. Chaining enables computation of the key for all nodes in a large DAG without executing any parts of the DAG.

In the next section (§III-C2), we describe a case where the system falls back to assigning DAG nodes integer based IDs

(i.e. not KeyChain). All DAG nodes that receive input from nodes with integer IDs must also be assigned integer IDs and are not able to use KeyChain because integer IDs are not valid across programs (§II-B).

2) Outside Data: Origin Descriptors: The string representation of input data needs to encode when the input data last changed, so that out-of-date cached data will not be referenced. For instance, when a file is cached and the file contents change, a different key should be used to search for file so that old data is not referenced. Prior systems such as Nectar and Incoop accomplish this by using a hash of the data as the representation, which ensures the representation reflects the latest data, but significant I/O costs will be incurred to read large inputs (although the filesystem can be modified to compute this hash, as in Incoop).

Instead, KeyChain uses an origin descriptor (OD) to represent outside data. ODs should: (1) *describe where the data came from* (i.e. the file path) and (2) *include when the data was last changed, or the timeframe it is valid for* (i.e. the file's last modified time). Since the OD is computed whenever the key is computed, the key will reference the up to date data.

The exact way ODs are computed depends on the metadata available for a particular storage system. When the storage system does not have metadata about when data was last modified, the user could specify a time window for acceptably up-to-date data. For instance, a database query of total revenue might be acceptably up to date so long as it includes yesterday's sales, and this could be reflected in the OD as: '`<query string> <current day/month/year>`'. If users of different programs read the same data, but disagree on the acceptably up-to-date value, their programs will compute different keys and not share data.

If neither of the above approaches are possible because the storage system doesn't provide a last modified time and the program must read the most up-to-date data, this is a sign that the program and data it produces might not be suitable for memoization. In this case, we should also avoid hashing input. When KeyChain detects that there is no OD, it falls back to simple integer IDs as described in §II-B.

The one advantage that hashing data brings over the above techniques is that hashing data can ensure programs will share data even when they read from logically different places, such as duplicate files. Incoop relies on the storage system hashing stored data. KeyChain does not require that data is hashed, but could use a hash as the origin descriptor, if present.

3) Random Input Data: Random data can be generated two ways: by input nodes, or by a UDF in a transformation node. Generating random data in a UDF introduces non-determinism and is usually avoided when writing data processing programs (§II-A,III-D). However, random data is often generated by an input node, such as when initializing machine learning algorithms or generating test data. Since pseudo random number generators are deterministic with respect to their parameters [21], KeyChain assigns these nodes an OD that includes those parameters, such as the name of the algorithm, the seed, and the size of data generated.

D. KeyChain Assumptions and Limitations

Some limitations of KeyChain stem from limitations of UDF hashing: (1) Programs compiled by different compilers that produce different output will never match. (2) There is no guarantee that functionally equivalent programs will produce the same hash. If a compiler produces different output for equivalent programs that have minor syntax differences, then data can only be shared between UDFs that have exactly the same syntax. These limitations don't cause fundamental issues for CPM (sharing is still possible), but it is ideal to share data between programs that are *functionally equivalent* even if they are not syntactically identical. We investigate this further in §V-C.

KeyChain assumes that it is possible to access a hash of UDF bytecode. Bytecode is easily available at runtime from the JVM, but we are not aware of this feature in other languages/runtimes aside from the experimental pass we developed for LLVM. We hope that this paper leads to interest in adding such capability to other systems.

As discussed in §II-A, KeyChain assumes that UDFs are deterministic: that a program's output is determined by its source code and input. KeyChain does not add any program requirements, because this assumption is already made by Spark, MapReduce and others to enable fault tolerance.

KeyChain does not enforce access control. An attacker could compute a key for a file path they cannot access and try to read the data from a cache. Enforcing access control on cached data is challenging as access control is very deployment dependent and is outside the scope of this paper. But, one way to enforce access control to cached data would be to ensure it can only be read by users that can access the original data. If a storage system is unable to enforce such access control, the cached data could be encrypted, and the encryption key(s) made available to those with the proper access rights.

IV. KeyChain in Apache Spark

This section describes our implementation of KeyChain in Apache Spark 2.2. Since one contribution of this work is showing the simplicity of a CPM implementation, we describe our implementation, some challenges we encountered, and what made it simple. Overall, we wrote under 2100 lines of code, including a UDF hashing library for the JVM (<1000 lines) (§IV-A) and modifications to Spark's execution layer to enable KeyChain (§IV-B) and transfer data between Spark contexts (§IV-C) (<1100 lines). None of the changes require modifications to existing Spark programs. Our changes to Spark are publicly available[6].

A. UDF Hashing Library

We wrote a UDF hashing library in Scala, and have made it publicly available[7].

Usage. The library provides `Hash(func)`, which accepts a reference to a JVM function and returns a string containing

[6]https://github.com/craiig/spark-keychain/tree/branch-2.2-keychain

[7]https://github.com/craiig/keychain-tools/tree/master/udf-hash

its hash. When a node is added to the DAG, Spark passes the node's UDF to `Hash` and stores the result.

Implementation. `Hash` uses the ASM analysis framework [22] to read the bytecode of the function and its callees. Class reflection [23] is used to find and hash all referenced variables. `Hash` applies filtering to remove unnecessary details from ASM's output (below), and passes the remainder to a cryptographic hash (SHA256). To improve performance, `Hash` caches previously hashed function bytecode (but not variable values).

Challenges. The largest challenge in using the UDF hashing library is ensuring that hashes don't needlessly vary. One way this happens is that Spark assigns a random UUID [24] to each JVM and tags all SQL expressions with it, making otherwise equivalent UDFs produce different hashes. To find such problems, `Hash` can output a JSON trace and our library includes tools to compare traces. We used traces to identify the UUID issue, and added filters to exclude hashing the UUID.

B. Execution Layer Modifications

Key computation. Since Spark implements the DAG model, it was simple to add KeyChain to the execution layer. Spark already has the information needed for KeyChain to compute keys and the program structure to chain keys from input nodes. *Communication pattern* strings were taken directly from Spark's names for DAG operations. *Origin Descriptors* were added for input nodes that read from HDFS and local files. *UDFs* were hashed using our library.

Using keys to identify data. As described in §II-B and shown in fig. 1, Spark's execution layer indexes the cache using the integer ID of each DAG node to be materialized. Instead, we modified Spark to use keys generated by KeyChain. We retained the ability to use numeric IDs for cases where we could not compute or did not implement an origin descriptor (§III-C2). Spark computes the IDs for every DAG node once so it does not add lots of extra computation to call KeyChain to generate keys. Overall, to support the entirety of TPC-DS, we added KeyChain to 26 different DAG node types in Spark, with each change averaging 14 lines.

Handling complex operations Our treatment of Spark's shuffle operation highlights how simple adding KeyChain can be, even with complex code. Shuffle is an all-to-all communication primitive used to implement reduce, group by, and join operations. Shuffle requires the movement of many data blocks between worker nodes, and each block is tracked using a numeric ID. We could have changed the shuffle implementation to use KeyChain keys for shuffle block IDs, but this was not necessary because only the final result of the shuffle is useful to cache. Instead of deeply modifying the shuffle implementation, we just used KeyChain to compute a key for the final output.

C. Cross Instance Sharing

Users often share a *single* Spark context, allowing them to share data that is cached in the same process, but users also use *separate* contexts for resource isolation. To enable sharing between separate contexts, we modified Spark so contexts can query one-another for cached data. So that applications do not inadvertently share data identified by numeric IDs, only KeyChain keys are requested from other contexts. (§II-B). In our current implementation, contexts are manually linked via an API call and no access control checks are performed when sending cached data to another context. Sharing between contexts requires extra computation to serialize and transfer the data to another context (§IV-D), evaluated in §V-B.

D. KeyChain in Spark Limitations

The largest issue with adding CPM (via KeyChain) to Spark is that the lifetime of cached data can be short. Data in Spark's in-memory cache is managed by an application (driver in Spark terminology), so cached data is lost when an application shuts down. However, applications can be long lived when they are used interactively, such as when Spark is used in a shared notebook system like Jupyter [25] or Zeppelin [26]. To extend the lifetime, cached data could be stored on external storage such as HDFS [27] or Alluxio [28] with KeyChain being used to compute the filenames, so data can be found by later applications.

Like Spark itself, we rely on the programmer (or a higher level library) to mark useful results to cache. Prior work has shown how to automatically decide what to cache, and even how to re-write programs to achieve better sharing, so we did not investigate these issues in this work (§VI-3).

The main drawback of reading cached data from other Spark instances or from a network file system is that serialization and deserialization overheads can greatly impact the speedup achievable from sharing (§V-B). Serialization overhead in Spark has been investigated by Ousterhout et al. [29] and Neutrino [12], and Skyway [30] presents a way to reduce JVM serialization overheads.

V. Evaluation

First, we show a situation where data sharing occurs to show how CPM (as implemented by KeyChain) can help to achieve more data sharing and high speedups thanks to the re-use of cached data (§V-B). Next, we show how UDF hashing can identify some functionally equivalent programs, so that sharing can occur in more situations (§V-C). Finally, we show that the overheads of KeyChain are negligible so it can be safely enabled even when data sharing may not occur and systems and users can benefit from CPM when sharing does occur (§V-D).

A. Experimental Setup

Test machines. We ran the tests in §V-B and V-D1 on a machine with two 2.6 Ghz AMD Six Core Opteron 2435 processor with 32GB of RAM with a 160GB 7200rpm HDD used to store Spark shuffle data. For our larger scale tests in §V-D2, we used Microsoft Azure, described later.

Spark. We tested an unmodified version of Spark 2.2 and our KeyChain implementation in Spark. For brevity in this section, we call our modified version KeyChain. We setup Spark to use

all cores and all available memory. For our interprocess tests in §V-B, each instance used half the available memory.

Benchmarks and Test Data. For a source of realistic data and queries, our tests use TPC-DS [13, 14], a benchmark designed to represent modern decision support tasks, including ad-hoc and reporting queries. TPC-DS includes query and data generators. We used the TPC-DS benchmarking framework from Databricks [31] configured to generate and use Parquet [32] format data.

Avoiding JVM bias. Most of our tests ran on the JVM, which has a just-in-time compiler (JIT) that optimizes the program as it runs and a garbage collector that runs across allocated memory. To ensure consistent results (but not necessarily the best performance), we restart the JVM after each test so that later tests do not benefit from prior tests warming up the JIT optimizer [33, 34] and so that later tests are not hindered by the garbage collector cleaning up memory allocations performed by earlier tests. For our TPC-DS tests, we restarted the JVM after each full run of all queries.

B. How KeyChain Enables More Sharing

To highlight the gains that CPM can achieve through sharing data more widely, we perform a series of experiments to mimic an interactive use of Apache Spark in a context where data sharing occurs. Returning to the example from the introduction, we simulate Alice and Bobbie beginning to explore a data-set and running the same tasks: (1) *scanning* the TPC-DS `store_sales` table and caching it in memory for faster access, and (2) *joining* `store_sales` to the `customer` table and then caching those results in memory. Both tasks are written using the Spark Dataframe API. Our tests were run on a 2GB dataset on a Spark instance with 13GB in-memory cache.

Performance improvements of caching. Table I shows the overall performance of Alice and Bobbie running both tasks. Alice runs her query first and the data is loaded and cached in memory (Cache Miss in table I). When Bobbie runs the same tasks, they re-use the data loaded by Alice (Cache Hit) leading to a 189x and 281x speedup for each task.

The above speedups are high because data is re-used from the same process without serialization or data transfer overhead. Table II shows the performance when data is serialized in memory and transferred to another process on the same machine. When serialized data is read from the same process, speedups drop to 2.6x-4.4x, and to 2.2x-3.6x when serialized data is transferred to another process. High serialization overheads are endemic to Spark, and not due to anything added by KeyChain. Not all systems will have such high interprocess overhead.

KeyChain's increased sharing potential. Now, we look at **if** sharing will occur when Alice and Bobbie run their tasks in different sharing scenarios, and how CPM (as implemented by KeyChain) can achieve more cache hits than unmodified Spark. One way that these scenarios occur in practice is when users are using interactive notebooks like Jupyter [25] or Zeppelin [26], which allows users to work collaboratively on the same code or just share a connection to a single Spark context. Table III shows the results. **Same Code** is when Alice and Bobbie are collaborating to write the same program and can reference each other's variables. *This is the only case in which unmodified Spark can make use of cached data.* **Same Process** and **Diff. Process** are when Alice and Bobbie write completely separate code, but that code is run in either the same or different process. **KeyChain achieves more cache hits than Spark because KeyChain enables cross program memoization.**

TABLE I
PERFORMANCE OF CACHE HITS AND MISSES FOR BOTH EXAMPLE QUERIES WHERE THE DATA IS SHARED WITHIN A PROCESS.

Shared Process	Scan	Join
Cache Miss	39±4.8s	101±2s
Cache Hit	0.21±0.02s	0.36±0.02s
Speedup	189x	281x

TABLE II
PERFORMANCE OF CACHE HITS AND MISSES FOR BOTH EXAMPLE QUERIES WHERE THE DATA IS *shared between processes*.

Interprocess (Serialized)	Scan (s)	Join (s)
Cache Miss	151±6s	98±3s
Cache Hit	56±2	22±1
Cache Hit w/ Data Xfer	69±6s	27s±1.1s
Hit Speedup	2.6x	4.4x
Hit w/ Data Xfer Speedup	2.2x	3.6x

TABLE III
CACHE HITS AND MISS ACHIEVED IN DIFFERENT SHARING SCENARIOS.

Same Code	Spark	Hit	Hit
	KeyChain	Hit	Hit
Same Process	Spark	Miss	Miss
	KeyChain	**Hit**	**Hit**
Diff. Process	Spark	Miss	Miss
	KeyChain	**Hit**	**Hit**

C. Syntactic Resilience Evaluation

This section investigates how compilers transform syntactically different but functionally equivalent input programs into the same output, a property we call *syntactic resilience*. If the compiler produces the same output for equivalent programs, then the UDF hash is the same, and computation can be shared *across* equivalent programs written by different users. Our goals in studying syntactic resilience is to: (RQ1) justify our choice of hashing UDF bytecode versus using program source, (RQ2) understand the limitations of using compiler output to detect program equivalence, and (RQ3) measure the current syntactic resilience of different compilers.

1) Methodology: One way to measure syntactic resilience is to collect real-world programs, manually verify (or create) equivalent UDFs, and then check if a compiler produces identical outputs. However, this does not help predict whether or not this effect will generalize to other UDFs that are not in the test set. Therefore, we developed a test suite to test compiler's resilience to specific syntactic variations to better understand the limitations of each compiler and syntactic resilience in general.

To create our test suite, we surveyed several sources (described below) to create a corpus of 64 syntactic *variations* that can be applied to change a program's syntax without changing its functionality. For each variation, we derived a test case of at least two code snippets, where the variation has been

applied to the first to create the second. Table IV lists some of these variations and example test cases. We have made our benchmark publicly available[8].

We do not claim our set of variations is exhaustive, but we do cover a larger variety of non-functional syntactic differences than prior work. Prior evaluations of syntactic resilience [15, 16, 17, 35] tested variations that *change* a programs functionality, such as adding a postfix increment operator to every variable reference and biased their results by repeatedly testing the same variation sometimes thousands of times.

We briefly describe our sources, why we chose them, and how we derived variants from each. We have made our full methodology and source data available, with direct links provided as footnotes on each source.

Source 1: TPC-DS[9] (7 test cases). To identify variations that can appear in data processing programs, we surveyed all expressions from TPC-DS and considered ways to re-write them while still retaining functional equivalence, and then added those variations to the corpus. For instance, if an expression included a commutative binary operator, then the operands could be swapped, which resulted in us adding the `Swap Operands` variation to the corpus.

Source 2: Compiler Documentation[10] (41 test cases). Since syntactic resilience is dependent on a compiler's optimizations, we exhaustively surveyed LLVM and GCC's optimizations and canonicalizations [36, 37]. When an optimization would transform equivalent code into the same output, such as LLVM and GCC's tree reassociation pass that canonicalizes mathematical expressions, we included a variation to test that optimization. Compiler optimizations are not guaranteed to produce the *same* output as a hand optimized version, but we find many do. We chose not to include some optimizations when the resulting test case would simply compare an unoptimized version to a hand optimized version that we felt was unrealistic for a programmer to write, such as auto-vectorization or jump-threading.

Source 3: Related Work[11] (11 test cases). TCE [15] tested against a set of manually verified equivalent programs taken from real world code. We grouped these equivalent programs into 11 test cases that tested distinct variations. We excluded TCE test cases that overlapped with existing tests in our corpus, but kept tests that were more complex than our prior test cases to evaluate real-world examples.

Source 4: Miscellaneous. The remaining 5 tests in our suite include a test for resilience to whitespace, `if(x!=0)` equivalence to `if(x)`, associative constant folding, pointer dereference equivalence to array indexing, and a for-loop with a single iteration.

Studied Compilers. We evaluate several compilers because while Java and Scala are used by Apache Spark, this is not the only choice for data processing systems. Data processing systems that compile to native code can outperform JVM-based systems [38, 39], so we test GCC and Clang (LLVM) to see if they are a better choice for equivalence checking. To ensure the code snippets were consistent in each compiler, we ported tests to each language with template-based code generation. Scala lacks a post- and prefix increment operators, so our Scala test skips those tests. To measure resilience, we compile each variant with each compiler and hash the output. For Java and Scala we use the UDF hashing library from KeyChain (§IV-A). For Clang/GCC output we hash the assembly output.

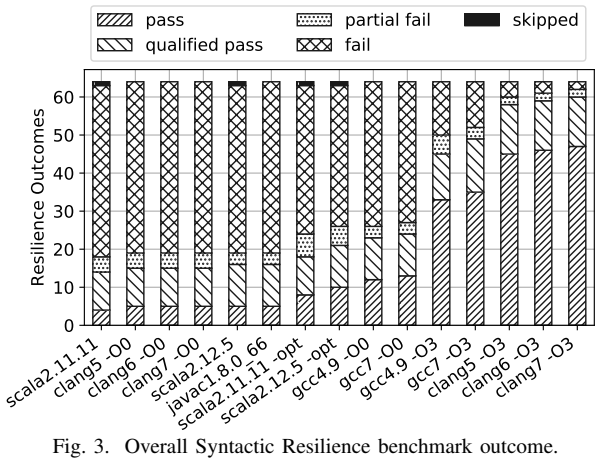

Fig. 3. Overall Syntactic Resilience benchmark outcome.

2) Results: We classify how well a compiler successfully performs on each test based on the number of unique outputs relative to other compilers. **Pass (P)** is when a compiler produces *one* unique output from a number of unique input programs. **Qualified Pass (QP)** is when a compiler produces more than one unique output, but no other compiler produces fewer. Qualified passes ensures we don't penalize compilers for difficult tests when no other compilers can do better. **Partial Fail (PF)** is when a compiler produces fewer unique outputs than it has inputs, but does not achieve a pass or qualified pass. **Fail (F)** is when a compiler produces the same number of unique outputs as unique inputs.

RQ1: Is hashing bytecode an improvement over program source? Table IV presents selected results from our benchmark to highlight common program variations. The **Whitespace** row shows *all compilers can provide basic resilience to whitespace, comments and variable name differences, an improvement over using program source.*

RQ2: What are the limits of compiler-assisted equivalence? All compilers fail Logical Operand Swap (LOS) because, strictly speaking, this variation is not functionally equivalent. Short circuit evaluation for logical-or means operand ordering changes execution behaviour, even though the order may be irrelevant. LOS highlights the limits of syntactic resilience: *when syntax variations imply changes in execution behaviour, these differences are not optimized away by the compiler.* This applies to memory allocations, side effects, early exit conditions, and more.

RQ3: What is the syntactic resilience of different compilers? Fig. 3 shows the aggregate results for each compiler.

[8]https://github.com/craiig/keychain-tools/tree/master/resilience_eval
[9]https://github.com/craiig/keychain-tools/tree/master/tpcds_analysis
[10]https://github.com/craiig/keychain-tools/tree/master/compiler_analysis
[11]https://github.com/craiig/keychain-tools/tree/master/tce_analysis

TABLE IV
SELECTED RESULTS FROM SYNTACTIC RESILIENCE BENCHMARKS.

Name and Description	Example(s)	Scala 2.12	Java 1.8	Clang 7	GCC 7
		Optimizations Off / On (where applicable)			

Whitespace
Add, change, remove whitespace, comments, variable names
	`(x+y)`
	`( x + y )`
	`( x + /*hello*/ y )`
	`(w+z)`
		P/P	**P**	**P/P**	**P/P**

Swap Operands Swap the operands of a binary commutative operator	`x+y -> y+x`	F/F	F	F/**P**	F/F
Logical Operand Swap Swap the operands of a logical comparison. Short circuiting in language specifications disallow treating these statements as equivalent.	`x || y` `y || x`	F/F	F	F/F	F/F

Constant Folding
Replace a constant with an equivalent expression
	`x+2`
	`x+1+1`
	`x+(1+1)`
	`1+1+x`
		PF/PF	PF	PF/**P**	**P/P**

Algebraic Tree Reassociation
Reassociate commutative expressions to promote better constant propagation [36]
	`(x+y)-(z+v)`
	`((x+y)-z)-v`
	`(x+y-z)-v`
	`x+y-z-v`
		PF/PF	PF	PF/**P**	PF/PF

Comparison Invert Apply De Morgan's laws to a comparison operation	`(x==1) && (y==2)` `!((x!=1) || (y!=2))`	**P/P**	**P**	PF/**P**	**P/P**

Comparison Swap
Invert the comparison statement and swap the basic blocks of an if-statement (*Passes on GCC 4.9 -O3)
	`if(x){return y;}`
	`else{return x;}`
	`if(!x){return x;}`
	`else{return y;}`
		F/F	F	F/**P**	F/F*

Loop Invariant Hoisting
Move a loop invariant from the inside of a loop. All tests are qualified pass as no compilers canonicalize to our hand hoisted version.
	`int a = 0;`
	`for(...){ if(x){ a=y; } ... }`
	
	`int a = 0;`
	`if(x && loop_condition)`
	`{ a = y;}`
	`for(...){ ... }`
		QP	QP	QP	QP

Compilers that perform more ahead of time optimizations are better at canonicalizing functionally equivalent code. Clang and GCC include many ahead of time optimizations and are the state-of-the-art for syntactic resilience. Scala and Java have less ahead of time optimizations because they rely on JIT optimization , but Scala with ahead of time optimizations (`-opt`) performs better than without. We also found GCC and Clang could be improved: most failures were due to variations in label names, register allocation, and ordering of independent instructions.

Overall our tests show that compiler canonicalization and optimization passes can be used to detect some equivalent programs. KeyChain leverages this effect to achieve more data sharing. While the syntactic resilience of Scala and Java lags behind GCC and Clang, finding equivalent programs by hashing JVM bytecode is more likely than using program source or simple integers (§II-B) to identify UDF computations, and our test suite can help guide future improvements.

D. Overheads

Now we examine the overheads of using KeyChain to implement CPM in Spark. We first measure the standalone performance of the UDF hashing library (§V-D1) and then the overall impact on TPC-DS when there is no potential for sharing (§V-D2).

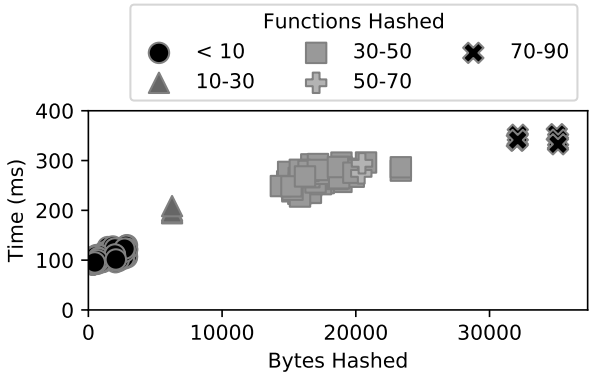

Fig. 4. Overall time to hash Java and Scala functions from our resilience benchmark using the UDF hashing library.

1) UDF Hashing: Fig 4 shows the time to hash each of our Java and Scala syntactic resilience tests (§V-C) . The time to hash a UDF is determined by the number of bytes and functions hashed. The largest UDF takes under 350 milliseconds, while the smallest takes under 150 milliseconds. This shows UDF hashing performance in the conservative case, because each test is hashed using a newly created JVM with no cached results.

Table V shows UDF[12] hashing time for TPC-DS queries during the tests in (§V-D2). We also show performance when

[12]The UDFs in this case are Spark's database operators.

debug tracing is active and logging all hashing traces to disk, which is only used to debug UDF hashing (§IV-A). It takes 18 seconds to hash all UDFs used in TPC-DS, for an average of 2ms per UDF. There are over 9000 UDFs because Spark's operators are implemented as UDFs (§II-A). Compared to our prior results with the UDF hashing library, UDF hashing in a realistic scenario is much faster due to caching previously hashed functions and a warmed-up JVM.

TABLE V
UDF HASHING OVERHEAD ON TPC-DS. DURING *Normal* OPERATION, AND *Debug* WHEN UDF TRACING IS ENABLED FOR DEBUGGING.

Mode	Max	Avg	Min	Sum	Total
Normal	265ms	2ms	0.07ms	18s	9,345
Debug Miss	322ms	65ms	4ms	1.1s	17
Debug Hit	1.5s	11ms	0.1ms	108s	9,328

2) End to End Spark Performance: To show that KeyChain allows CPM to be enabled in all deployments without incurring high overheads, we compare the performance of TPC-DS with and without KeyChain when there is no sharing. In both cases, the empty cache is checked for relevant results. In the original Spark, the cache is indexed with integers. In the KeyChain version, the cache is indexed with KeyChain keys. Tests were run on large and small scale clusters on Microsoft Azure. Machine details and TPC-DS parameters are in table VI. HDFS was used to store data. TPC-DS includes 104 queries, but we disabled query 72 on Large due to too much shuffle data for some nodes, and 77 due to a memory leak when over 40,000 tasks were generated.

TABLE VI
MACHINE AND TPC-DS CONFIGURATION FOR END TO END TESTS.

Name	Small	Large
TPC-DS Scale Factor	10	1000
Partitions	50	160
Masters:Workers	1:3	1:20
Instance Type	D4S_v3	E8_v3
vCPUs per node	4	8
Memory per node	16GB	60GB
Storage per node	32GB	200GB

TABLE VII
PERFORMANCE OF TPC-DS. AVERAGE TOTAL COMPLETION TIME (*ATCT*) IS THE AVERAGE TOTAL RUNTIME OF EACH TPC-DS RUN.

	Small		Large	
	Spark	KeyChain	Spark	KeyChain
Iterations	5	5	3	3
ATCT (s)	3628	3684	34,328	33,732
Std.Dev.(s)	75	83	1002	857
	2%	2.2%	2.9%	2.5%

Table VII shows the results of our tests. Overall, we find that KeyChain has a negligible impact when there is no sharing, because the performance is within the standard deviation of the original Spark. We estimate that sources of system jitter like garbage collection [40], JIT compilation [33, 34], and stragglers [18, 41] make a larger impact than KeyChain. While our CPM implementation is decidedly simple, we have shown that KeyChain can implement CPM without incurring high run-time overheads when sharing is not possible.

VI. RELATED WORK

1) Memoization: Early work on memoization investigated how caching can help *single programs* [42] and what kinds of programs are suitable for caching [43]. KeyChain applies memoization *between* data processing programs.

2) Explicit Sharing and Incremental Processing Systems: When programs are *known* to benefit from prior results, programmers will write them to explicitly share data. For instance, Spark users often use Alluxio [28][13], an in-memory cache for distributed file systems, to store data they know can be shared. If a program can re-use its own results, incremental processing frameworks can be used. For instance, Incoop reports overheads of 5-22% for initial runs with no cached data, but this pays off with 3-1000x run time improvement on subsequent runs [5]. However, explicit sharing or incremental processing depends on individual programmers knowing about a sharing opportunity ahead of time. Low-overhead techniques to identify results of computation like KeyChain are useful when opportunities for sharing are *not guaranteed* or might be difficult for programmers to identify.

Nectar [1] uses similar techniques to KeyChain and reported a 20-50% reduction in computation time on Microsoft's clusters due to CPM. Nectar hashes C# bytecode but also *hashes input data*, a source of overhead that KeyChain avoids. Nectar does not report overhead, implementation complexity, or syntactic resilience of the hashing technique and their implementation is not publicly available. This paper contributes an improved design with negligible overhead so that CPM can be deployed more widely, and a evaluation of syntactic resilience that tests the limits of UDF hashing.

3) Materialized view selection and cache placement algorithms: Materialized view selection techniques analyze past SQL queries to determine subexpressions to cache for the benefit of future queries, but do not consider UDFs [3, 7, 8] or if equivalent UDFs can be shared between programs [9, 10]. Spark's SQL caching layer will search for relevant cached SQL expressions, but not UDFs and does not work across Spark instances. KeyChain computes identifiers for the results of UDF computations that these techniques can use to decide what is beneficial to cache [11, 12]. For instance, [3] shows that they can save up to 40% of machine hours in their cluster. However, these benefits can only be realized once the common subexpressions can be identified.

4) Value Numbering in Compilers: Value Numbering (VN) [45] is a compiler pass that eliminates redundant expressions by computing identifiers for expressions in a basic block by hashing operands and operators. KeyChain can be seen as applying ideas from VN to the data processing setting.

5) Program Equivalence Detection: UDF hashing is a heuristic equivalence detection technique. Program equivalence is undecidable [6] so feasible approaches are either provers or heuristics. Provers, such as Cosette [46], search for a series of valid transformations to transform one program to another, but are not ideal for low overhead equivalence checking because they cannot guarantee termination and would need pair-wise comparisons over all programs with cached data. Clone detection [35] detects *similar* programs, but con-

[13]Formerly Tachyon [44]

siders programs with *different functionality* to be similar, which is not suitable for CPM. Trivial Compiler Equivalence (TCE) [15, 16, 17] eliminates redundant test cases by compiling and removing test cases with duplicate outputs. KeyChain uses this same effect to gain wider sharing potential, and our evaluation tests only functionally equivalent variations (§V-C1).

VII. Conclusion

This work showed that KeyChain enables cross program memoization implementations to achieve significant performance benefits when sharing occurs, while incurring negligible overheads when sharing is not possible. We hope these results encourage the addition of CPM to more data processing systems.

References

[1] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *Operating Systems Design and Implementation*. USENIX, 2010, pp. 75–88.

[2] S. Jain, D. Moritz, D. Halperin, B. Howe, and E. Lazowska, "Sqlshare: Results from a multi-year sql-as-a-service experiment," in *International Conference on Management of Data*. ACM, 2016, pp. 281–293.

[3] A. Jindal, K. Karanasos, S. Rao, and H. Patel, "Selecting subexpressions to materialize at datacenter scale," *Proc. VLDB Endow.*, vol. 11, no. 7, pp. 800–812, Mar. 2018.

[4] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence: an analysis of hadoop usage in scientific workloads," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 853–864, 2013.

[5] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin, "Incoop: Mapreduce for incremental computations," in *Proc. 2nd ACM Symposium on Cloud Computing (SOCC)*. ACM, 2011, pp. 7:1–7:14.

[6] B. A. Trakhtenbrot, "Impossibility of an algorithm for the decision problem in finite classes," *Proc. USSR Academy of Sciences*, vol. 70, pp. 569–572, 1950.

[7] M. Kunjir, B. Fain, K. Munagala, and S. Babu, "Robus: Fair cache allocation for data-parallel workloads," in *International Conference on Management of Data*. ACM, 2017, pp. 219–234.

[8] P. Roy, K. Ramamritham, S. Seshadri, P. Shenoy, and S. Sudarshan, "Don't trash your intermediate results, cache 'em," *CoRR*, vol. cs.DB/0003005, 2000.

[9] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey, "Opportunistic physical design for big data analytics," in *International Conference on Management of Data*. ACM, 2014, pp. 851–862.

[10] ——, "Miso: souping up big data query processing with a multistore system," in *International Conference on Management of Data*. ACM, 2014, pp. 1591–1602.

[11] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated memory caching for parallel jobs," in *Networked Systems Design and Implementation*. USENIX, 2012, pp. 20–20.

[12] E. Xu, M. Saxena, and L. Chiu, "Neutrino: Revisiting memory caching for iterative data analytics." in *HotStorage*. USENIX, 2016.

[13] R. O. Nambiar and M. Poess, "The making of TPC-DS," *Proc. VLDB Endow.*, pp. 1049–1058, 2006.

[14] M. Poess, B. Smith, L. Kollar, and P. Larson, "TPC-DS, taking decision support benchmarking to the next level," in *International Conference on Management of Data*. ACM, 2002, pp. 582–587.

[15] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *International Conference on Software Engineering*. IEEE, 2015, pp. 936–946.

[16] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Trans. Softw. Eng.*, 2017.

[17] M. Houshmand and S. Paydar, "TCE+: An extension of the tce method for detecting equivalent mutants in java programs," in *Fundamentals of Software Engineering (FSEN)*. Springer, 2017, pp. 164–179.

[18] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[19] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Networked Systems Design and Implementation*. USENIX, 2012, pp. 2–2.

[20] Y. Huai, "A Deep Dive into Spark SQL's Catalyst Optimizer," https://www.slideshare.net/databricks/a-deep-dive-into-spark-sqls-catalyst-optimizer-with-yin-huai.

[21] Wikipedia, "Pseudorandom number generator," https://en.wikipedia.org/wiki/Pseudorandom_number_generator, 2018.

[22] OW2 Consortium, "ASM," http://asm.ow2.org/.

[23] Oracle, "java.lang.Class API Documentation," https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html.

[24] Wikipedia, "Universally unique identifier," https://en.wikipedia.org/wiki/Universally_unique_identifier, 2018.

[25] "Jupyter Notebook," http://jupyter.org/.

[26] "Apache Zeppelin," https://zeppelin.apache.org/.

[27] "Apache Hadoop," http://hadoop.apache.org/.

[28] "Alluxio," http://www.alluxio.org/.

[29] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *Networked Systems Design and Implementation*. USENIX, 2015, pp. 293–307.

[30] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu, "Skyway: Connecting managed heaps in distributed big data systems," in *Architectural Support for Programming Languages and Operating Systems*. ACM, 2018, pp. 56–69.

[31] Databricks, "Spark SQL Performance Tests," https://github.com/databricks/spark-sql-perf, 2018.

[32] "Apache Parquet," http://parquet.apache.org/.

[33] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't get caught in the cold, warm-up your JVM: Understand and eliminate jvm warm-up overhead in data-parallel systems." in *OSDI*, 2016, pp. 383–400.

[34] E. Barrett, C. F. Bolz, R. Killick, V. Knight, S. Mount, and L. Tratt, "Virtual machine warmup blows hot and cold," *CoRR*, vol. abs/1602.00602, 2016.

[35] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *International Conference on Software Engineering*. ACM, 2016, pp. 1157–1168.

[36] "LLVM's Analysis and Transform Passes," http://llvm.org/docs/Passes.html.

[37] GCC, "Options That Control Optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[38] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proc. VLDB Endow.*, vol. 4, no. 9, pp. 539–550, Jun. 2011.

[39] G. Essertel, R. Tahboub, J. Decker, K. Brown, K. Olukotun, and T. Rompf, "Flare: Optimizing apache spark for scale-up architectures and medium-size data," in *Operating Systems Design and Implementation*. USENIX, 2018.

[40] M. Maas, T. Harris, K. Asanovic, and J. Kubiatowicz, "Trash day: Coordinating garbage collection in distributed systems," in *Hot Topics in Operating Systems*. USENIX, 2015, pp. 1–1.

[41] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Operating Systems Design and Implementation*. USENIX, 2008, pp. 29–42.

[42] D. Michie, "Memo functions and machine learning," *Nature*, vol. 218, no. 5136, pp. 19–22, 04 1968.

[43] J. Mostow and D. Cohen, "Automating program speedup by deciding what to cache," in *International Joint Conference on Artificial Intelligence*. Morgan Kaufmann Publishers Inc., 1985, pp. 165–172.

[44] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Symposium on Cloud Computing*. ACM, 2014, pp. 6:1–6:15.

[45] T. VanDrunen and A. L. Hosking, "Value-based partial redundancy elimination," in *Compiler Construction*, E. Duesterwald, Ed. Springer Berlin Heidelberg, 2004, pp. 167–184.

[46] S. Chu, K. Weitz, A. Cheung, and D. Suciu, "HoTTSQL: Proving query rewrites with univalent sql semantics," in *Programming Language Design and Implementation*. ACM, 2017, pp. 510–524.