Practical Cross Program Memoization with KeyChain

Craig Mustard and Alexandra Fedorova
University of British Columbia
Vancouver, BC, Canada

Best Paper at IEEE International Conference on Big Data 2018!

Memoization:

Save resources and improve response time by reusing previously computed results.

“Cross Program Memoization” (CPM):

Reuse results between programs.

Basic Memoization

```
1 cache = {}
2 def memoized_slow_func(param1, param2, param3):
3     lookup_key = compute_key(param1, param2, param3)
4     if lookup_key not in cache:
5         cache[lookup_key] = slow_func(param1, param2, param3)
6
7     return cache_lookup[lookup_key]
```

Memoization in general:

1. **Compute a key** representing the computation to be done **(line 3)**
2. **Check the cache** for that key **(line 4-5)**
3. Return results if found, compute if not found, and (optionally) cache it.
   a. Not shown: choosing what to cache, eviction policies,

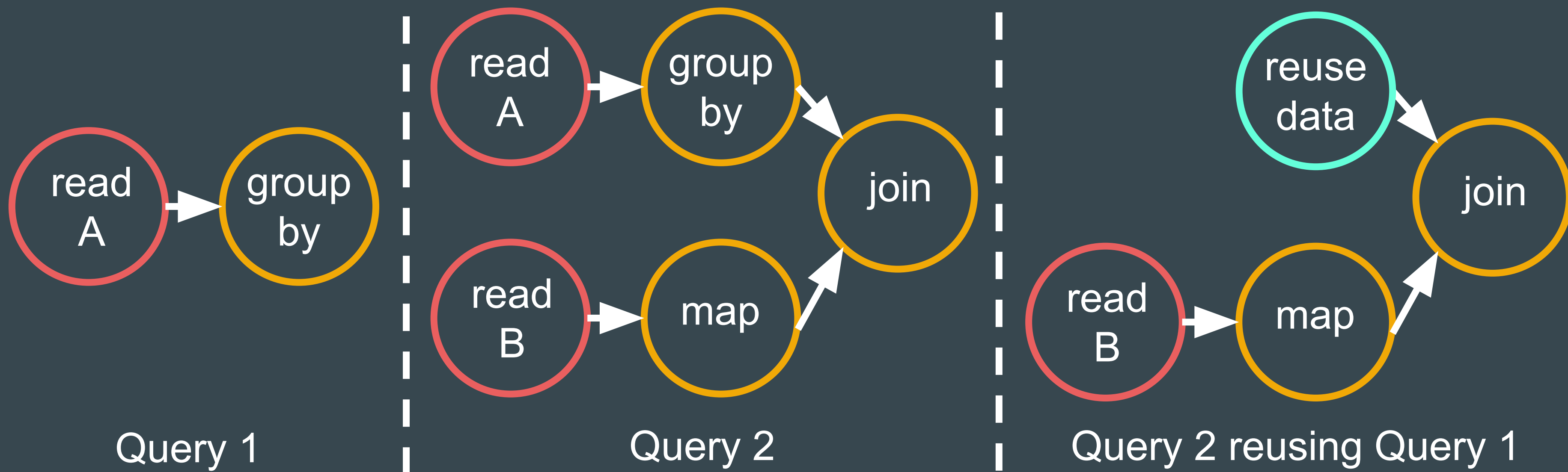
Cross Program Memoization

is possible when cache keys are valid across programs that share a cache.

This work shows how to effectively generate keys for intermediate and final results of data analytics programs.

Structure of Data Analytics Programs

- ➔ Programs are modeled as a directed acyclic graph (DAG)
- ➔ Gives an opportunity for finding prior results of parts of the program.



CPM is very effective, but only when the potential is there.

- ➔ CPM can be very effective:
  - ◆ 20-50% total machine-hours saved [Nectar, OSDI 2010]
  - ◆ 10-42% total machine-hours saved [BigSubs, VLDB 2018]
  - ◆ 37% reduction in query time [SQLShare, ICDM 2016]
- ➔ But potential for data sharing varies, and can be non-existent.
  - ◆ Only 10-20% on certain clusters [Nectar, BigSubs]
  - ◆ SQLShare queries benefit a lot (>90%) or a little (<10%)
  - ◆ Less than 1% of files are shared in academic clusters [Ren VLDB 2013]
- ➔ Want to enable CPM all the time, but need to address overheads when sharing potential is low.

When sharing potential is low, overhead is important.

- No sharing == no benefit from reusing results.
- The overhead of CPM becomes critical.
- Want to keep CPM enabled in case sharing potential changes.
- Prior work on CPM has not looked at overheads for low-sharing situations.
  - Nectar does not report overheads and Incoop can increase runtime by 5-22%

To have CPM always-on, we need low overhead techniques!

User-defined Functions:

- ➔ Data analytics systems, like Apache Spark, are powerful and general purpose thanks to user-defined functions, which are written in the same language as the data analytics system.
- ➔ Ideally, a CPM system would (heuristically) detect equivalent UDFs to share data between them, by computing the same key for both UDFs.
- ➔ Prior work has shown that compilers can be used to detect program equivalence.
  - ◆ If compilation(A) == compilation(B) then program A is equivalent to B.
  - ◆ [Trivial Compiler Equivalence, ICSE 2015]
- ➔ No one has investigated this effect in the context of CPM.

We show how to share data between (some) equivalent UDFs.

Challenge #1:

Sharing is not always possible, so we need to design low overhead CPM techniques.

Challenge #2:

Data produced by equivalent user-defined functions should be shared when possible.

Contributions: KeyChain

1. A simple to implement technique that computes keys for intermediate and final results that are valid across programs to enable CPM.
2. A low overhead design that computes keys in under 350ms, with negligible runtime overhead in practice. Overhead does not grow with data-set size.
3. Evaluation of compiler-assisted UDF equivalence with a new benchmark.
4. Implemented in Apache Spark 2.2
   a. Modifications and benchmarks are available online:
   b. https://github.com/craiig/spark-keychain
   c. https://github.com/craiig/keychain-tools

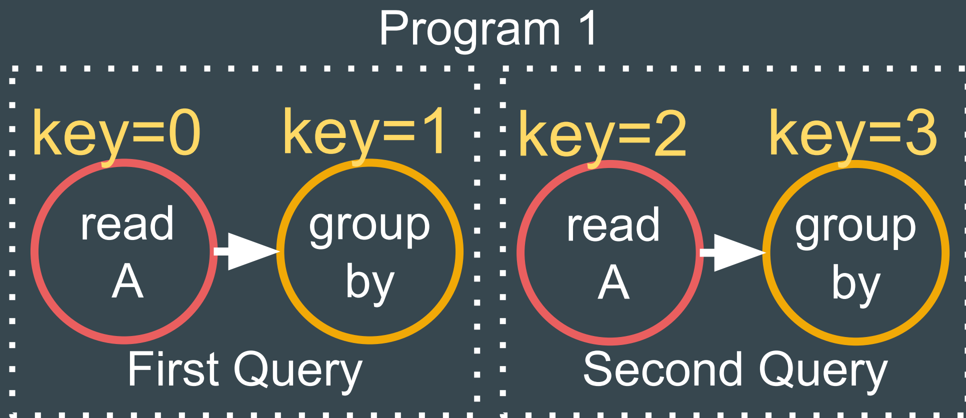
What KeyChain is not

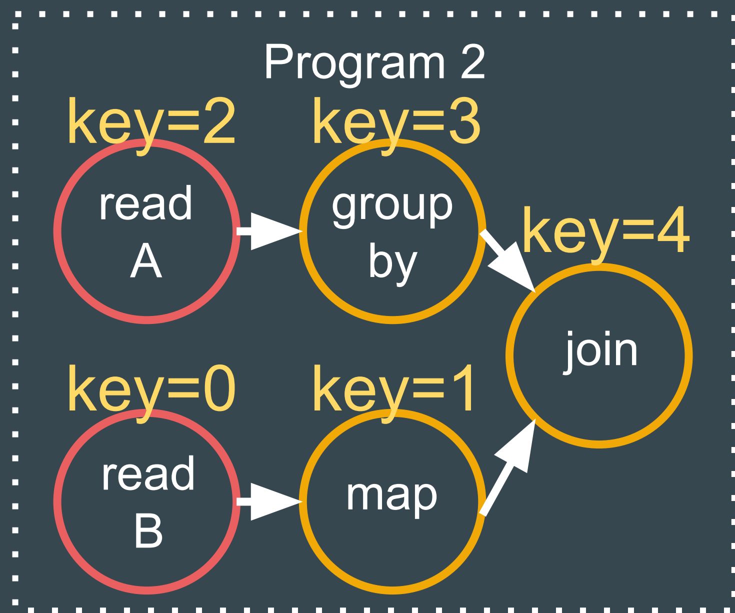
- **Not** a technique to decide what is beneficial to cache
  - Prior work considers potential reuse, computation/serialization/transfer costs.
    - Memoization in general [Michie 1968, Mostow 1985]
    - Databases: materialized view selection [ROBUS, MISO, BigSubs]
    - Data Processing: coordinated caching [Nectar, PACMan, Neutrino]
  - All techniques rely on having a key for the candidate data.
  - KeyChain computes a key so this prior work can be applied!

KeyChain Details

Motivating Example: Apache Spark

- Apache Spark implements user-managed caching, but not CPM.
- Code must directly reference prior results.
- Assigns per-program integers to cached data
- Results in two problems:



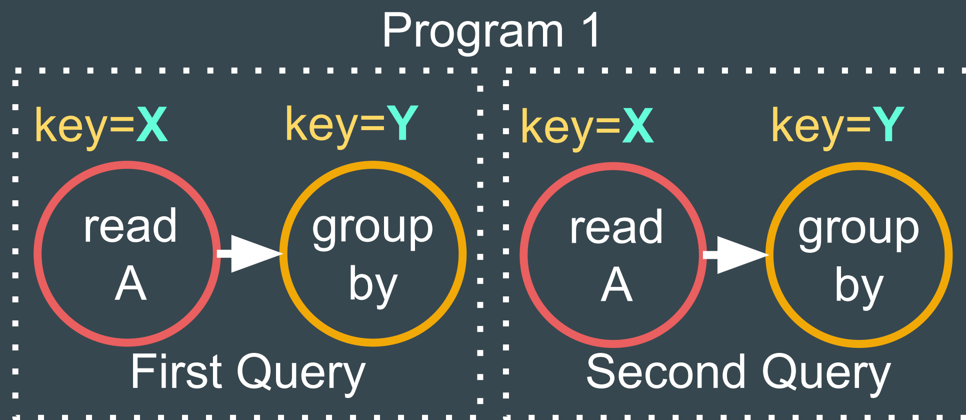


Keys are not reused within a program

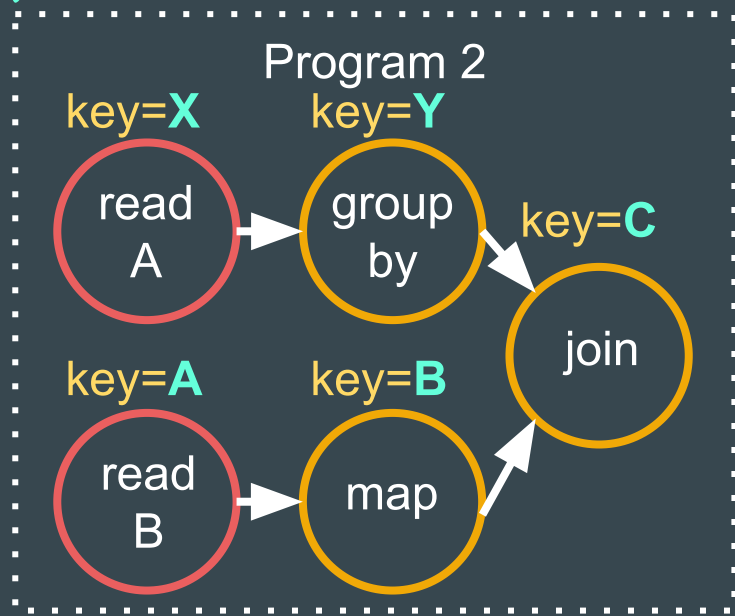
Keys not valid across programs

Solution: KeyChain uniquely identifies each node

- Keychain computes a key, KC_z, that is valid across programs.
- Equivalent computations compute the same key.



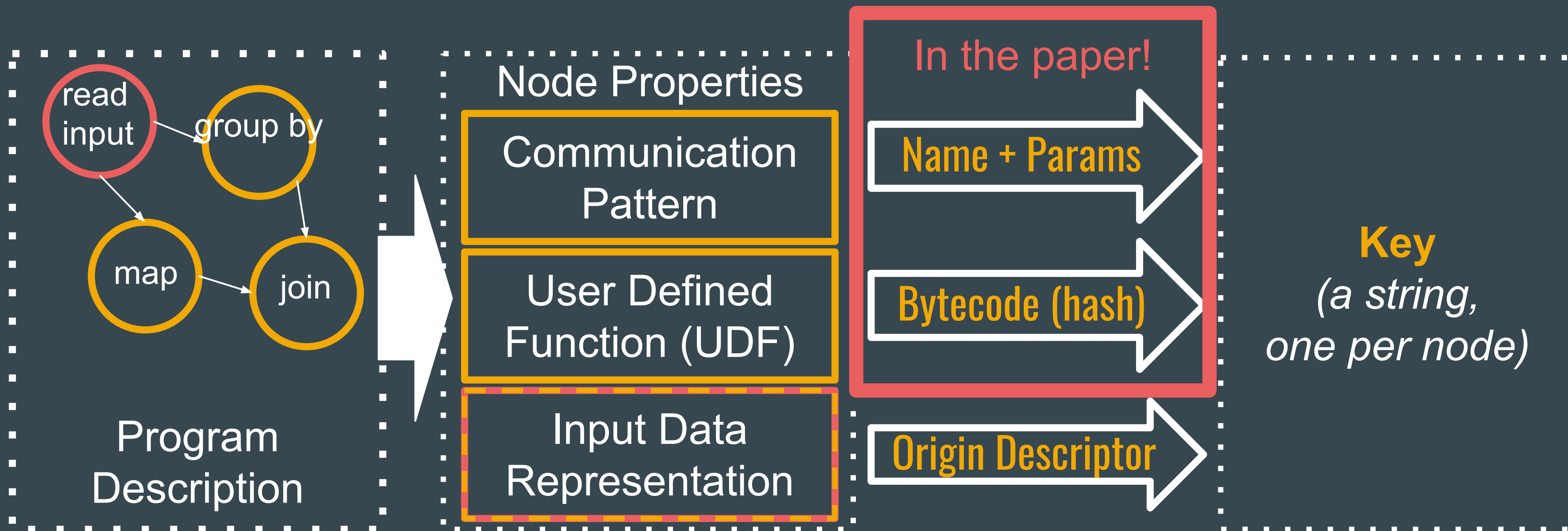
Keys reused within a program



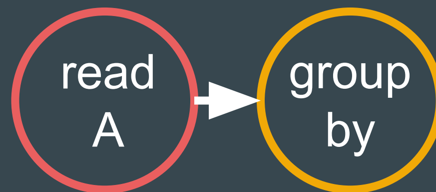
Keys are valid across programs

KeyChain Design:

- ➔ Goal: uniquely identify each node
- ➔ A key is a string that uniquely identifies each DAG node.
- ➔ Keys generated before the program is run or input data is read.



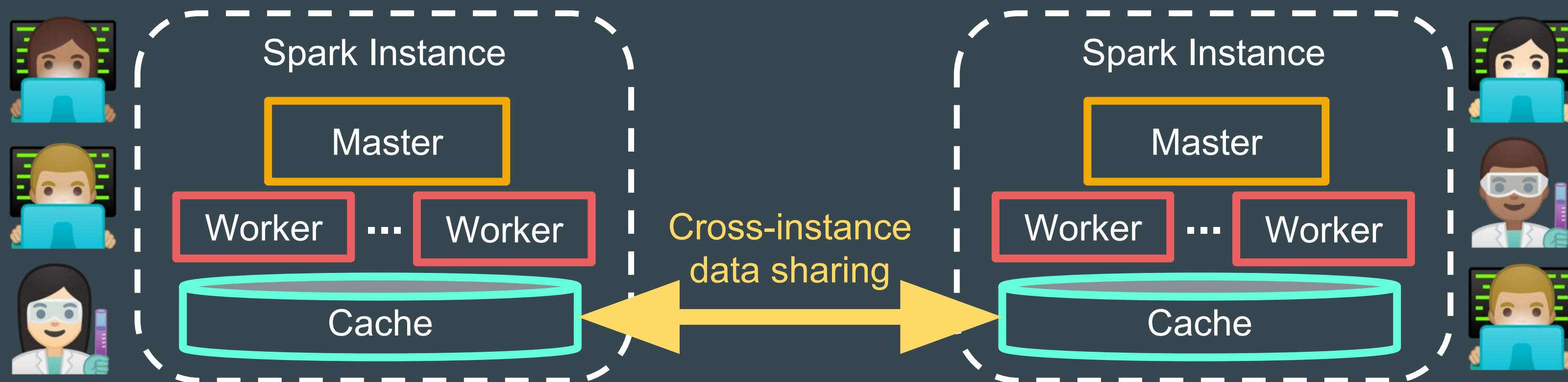
Input Representation: Origin Descriptors



- Need to represent input data as a unique string
- Prior work hashes input data [Nectar, Incoop]
  - A large source of overhead and data traffic that grows with data size
  - KeyChain avoids hashing input data.
- Our solution, Origin Descriptors:
  - Simply describe where the data came from
  - **And**: the last modified time OR the duration it is valid for
  - Examples:
    - “hdfs://path/to/your/data_nov-22-2018_12:32pm”
    - “select * from ... <database> (12:30pm-12:35pm)” (user needs to supply)
- Guaranteed never out of date because programs alway look for the right time.

Contribution #1: KeyChain is simple to implement

- KeyChain used to implement CPM in Apache Spark 2.2
  - 14 lines of code per operator on average (26 operators total)
- Uses our JVM UDF Hashing Library < 1000 LoC (Reusable)
- Also added cross-instance sharing (optional, not part of KeyChain)
- Total Spark changes: ~1100 LoC



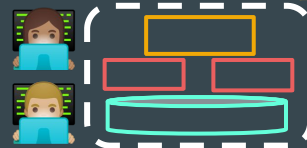
Summary: KeyChain in Spark enables more sharing

Assuming data has already been cached, when can we get a cache hit?

- ➔ When the same program can reference a prior result:

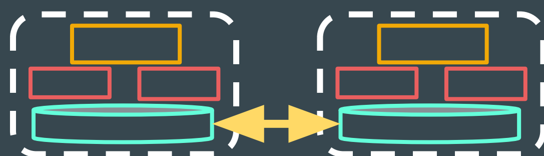
Apache Spark:	HIT	Spark + Keychain	HIT

- ➔ When same Spark instance independently computes the same result:



Apache Spark:	MISS	Spark + Keychain	HIT

- ➔ When different Spark instances independently computes the same result:



Apache Spark:	MISS	Spark + Keychain	HIT

Evaluation

- What are the performance benefits with more sharing in Apache Spark?
- What is the overhead of KeyChain when no sharing occurs?
- How well can UDF hashing detect equivalent code?

Evaluation: CPM Performance Benefits in Spark

Join Query	Cache Miss (s)	Cache Hit (s)	Speedup
Same Code Same Instance	100s	0.36s	**281x**
Different Instance	100s	27.0s	**3.6x**

Data movement costs

- Depending on the system, data movement can be an expensive operation.
- Data movement costs are specific to Spark, **not** to KeyChain.
- Ways to mitigate serialization overheads:
  - Optimize for the expected data re-use pattern [Neutrino, HotStorage 2016]
  - Avoid de/serialization with common format [Skyway, ASPLOS 2018]

Evaluation: CPM Performance Benefits in Spark

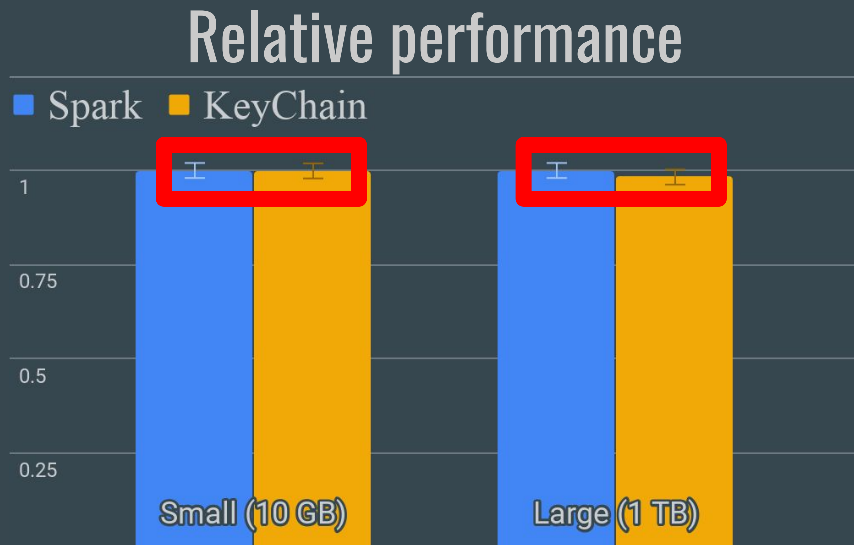
CPM with KeyChain yields significant speedup when sharing is possible.

Evaluation: End-to-end TPC-DS

- What are the overheads of KeyChain in practice?
- TPC-DS Evaluation on Microsoft Azure
  - Small - 4 machines each with 4 cores, 16GB RAM, Scale = 10 GB
  - Large - 21 machines each with 8 cores, 60GB RAM, Scale = 1 TB
  - NO CACHED DATA

KeyChain overheads are negligible in practice

Other sources of variation:
garbage collection, JIT compilation, stragglers, VM scheduling



Overhead does not grow with the data-set size.

Evaluation: UDF Hashing Overheads for JVM

- KeyChain Overheads: String concatenation + UDF Hashing

- Hashing equivalence test suite: < 350 ms
  - Cold JVM for each variant of test case
- Hashing in practice on all of TPC-DS:
  - Min: <0.1 ms / Mean: 2 ms / Max: 265 ms / Total: 18s (~9,000 UDFs)
  - Benefits from warmed up JVM and cached hashes (>99% high hit ratio)
- In general:
  - UDF hashing grows with the number of instructions hashed.
  - Does not grow with the data set size.

Evaluation: End-to-end TPC-DS

Contribution #2:

Low overheads lets us safely enable CPM
all the time!

Always benefit from potential sharing
with negligible overhead.

Evaluation: Can we detect equivalent UDFs?

- Test different compilers to inform CPM implementations (not just Spark)
- Contribution, new benchmark: 64 unique test cases, each with multiple variations.
  - Each case measures a specific type of code change, i.e. changing variable names
    - Pass if all variations compile to one unique output: compilation(A) == compilation(B)
  - Sources: TPC-DS, Compiler docs, Related work, Misc

Finding #1:
All compilers can detect equivalence when programs differ on only whitespace or variable names.
(Hashing bytecode better than program source.)

	Java 1.8	Scala 2.12-opt	GCC 4.9	GCC 7	LLVM 7
Full Passes	5	10	33	35	47
Qualified Passes	11	11	12	14	13

Finding #2:
Optimizing compilers are better at equivalence checking and being improved over time.

Finding #3: Fundamentally limited when syntax implies different execution behaviour
(i.e. evaluation ordering, early exits)

Evaluation: Can we detect equivalent UDFs?

Contribution #3:

Hashing bytecode helps to detect (some) equivalent UDFs!

Conclusion and Takeaways

(#1) Simple design makes KeyChain easy to add CPM to data processing systems.

(#2) KeyChain's low overhead means systems can always benefit from CPM, even when sharing potential is unknown or low.

(#3) Bytecode hashing helps to share data between (some) equivalent UDFs.

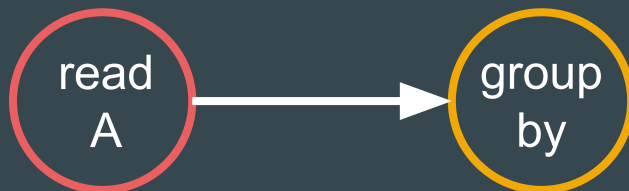
Modified Spark and benchmarks available online:
https://github.com/craiig/spark-keychain
https://github.com/craiig/keychain-tools

Backup Slides

Background: Structure of Data Processing Programs

- ➔ Many data processing systems model programs as a directed acyclic graphs (DAG)
- ➔ Input nodes and transformation nodes and data dependency edges

Input nodes read data from outside the computation model (HDFS, databases, etc)

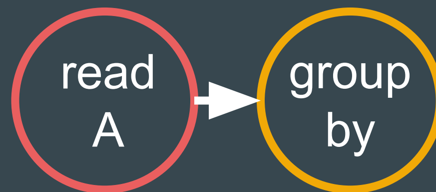


Data dependency edges model inputs and outputs of nodes

Transformation nodes represent operations on data.

Transformation node =
Input data (from nodes)
+ Communication Pattern: Map, reduce, group-by, join
+ User-defined Function How pattern affects data

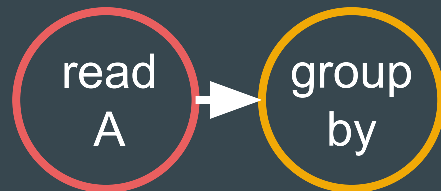
String Representations of Input Nodes



- ➔ Represent input data as a globally valid string
- ➔ Prior work hashes the data [Nectar, Incoop]
  - ◆ A large source of overhead and data traffic that grows with data size
- ➔ Our solution, Origin Descriptors:
  - ◆ Simply describe where the data came from
  - ◆ **And**: the last modified time OR the duration it is valid for
  - ◆ Examples:
    - “hdfs://path/to/your/data_nov-22-2018_12:32pm”
    - “select * from ... <database> (12:30pm-12:35pm)”
- ➔ Guaranteed never out of date because program always look for the right time.

- ➔ If it is not possible to give valid time range, is it may not be valid to cache

String Representations of Transformation Nodes



Compute key by concatenating:

- Input data:
  - The keys of any prior input nodes. (Chaining)
- Communication pattern: group-by, filter, map, etc
  - The string name of the pattern + parameters
- User Defined Function:
  - Assuming the UDF is deterministic (most are)
  - Any executable string: source code, bytecode, asm code
  - Best to hash UDF after compilation/optimization
    - JVM - Bytecode
    - C/C++ - Assembly Code or LLVM IR

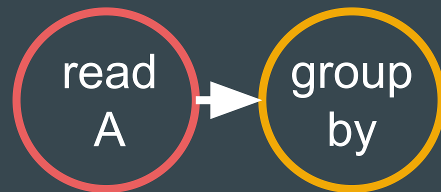
Node Properties

- Communication Pattern
- User Defined Function (UDF)
- Input Data Representation

Implementation: UDF Hashing Library

- Standalone UDF hashing library for the JVM: <1000 LoC
- `Hash(function) -> String`
  - Hashes all reachable bytecode and global values with SHA256
  - Caches results of previously hashed functions
- Challenge: Hashes vary due to reachable but irrelevant variables.
  - i.e. Spark assigns a Random UUID to each JVM instance
  - Solution:
    - Hashing Trace + Diff Tool: Easy to check what produced a given hash
    - Filters to ignore unnecessary variables
- On GitHub: https://github.com/craiig/keychain-tools/

Avoiding false positives:



Ensuring no false positives falls to CPM implementer who should have a good understanding of the systems.

+ Avoid CPM when there is a risk of false positives.
- Input data:
  - Understand the data source semantics to ensure proper details are included for each data source.
- Communication pattern:
  - Understand of the behaviour of communication patterns, which parameters change data results.
- User Defined Function Hashing:
  - Safe by default, but sometimes overly conservative
  - Filtering out the wrong variable leads to false positives.

Node Properties

- Communication Pattern
- User Defined Function (UDF)
- Input Data Representation

Handling data generators

Treat data generators as input nodes, with special Origin Descriptor.

Encode the parameters of the data generator as the Origin Descriptor

Instead of “hdfs://path/to/your/file”

We have: “prng=<W>_seed=SEED_dataWidth=32bytes_numItems=Z”